Software & Systems Modeling manuscript No.
(will be inserted by the editor)

Conformance Checking in UML Artifact-Centric Business Process Models

Montserrat Estañol · Jorge Munoz-Gama · Josep Carmona · Ernest Teniente

Accepted: May 2018

Abstract Business artifacts have appeared as a new paradigm to capture the information required for the complete execution and reasoning of a business process. Likewise, conformance checking is gaining popularity as a crucial technique that enables evaluating whether recorded executions of a process match its corresponding model. In this paper, conformance checking techniques are incorporated into a general framework to specify business artifacts. By relying on the expressive power of an artifact-centric specification, BAUML, which combines UML state and activity diagrams (among others), the problem of conformance checking can be mapped into the Petri Net formalism and its results be explained in terms of the original artifact-centric specification. In contrast to most existing approaches, ours incorporates data constraints into the Petri nets, thus achieving conformance results which are more precise. We have also implemented a plug-in, within the ProM framework, which is able to translate a BAUML into a Petri net to perform conformance checking. This shows the feasibility of our approach.

Published article can be downloaded from: https://doi.org/10.1007/s10270-018-0681-6

M. Estañol (✉) · J. Carmona · E. Teniente
Universitat Politècnica de Catalunya (UPC)
c/ Jordi Girona Salgado 1-3, 08034 Barcelona (Spain)
Tel.: +34 934 137 896
Fax: +34 934 137 833
E-mail: estanyol@essi.upc.edu (✉)
E-mail: jcarmona@cs.upc.edu
E-mail: teniente@essi.upc.edu

J. Munoz-Gama
Pontificia Universidad Católica de Chile (UC)
av/ Vicuña Mackenna 4860, Santiago (Chile)
E-mail: jmun@uc.cl

Keywords Conformance Checking · Artifact-Centric BPM · BAUML Framework · Process Mining

1 Introduction

Traditional approaches to business process modeling have mainly focused on representing the tasks that are part of the process and the order in which they should take place (i.e. the control flow perspective of the process). These tasks will likely require data, but its structure and how it is actually used by the tasks is not directly specified in the models. In contrast, the cornerstones of artifact-centric process models are the data definition and the details of what the tasks involved in the process do.

When business processes (either process or artifact-centric) are deployed, it may happen in practice that there is a gap between their definition and its implementation, i.e. processes are not exactly being executed in the way they were envisaged for. This might be due to errors made during development or deployment of the business processes, or because the users do not behave in the expected way.

Conformance checking is aimed at identifying such situations by detecting deviations between models and reality, and measuring the degree in which they differ [1]. However, current approaches for conformance checking have focused mainly on the control flow perspective [36, 5, 24]. Although useful, these results may not be as accurate as if data is taken into consideration, something that can be more naturally achieved by artifact-centric approaches.

As an example, consider a a city-bicycle rental system such as Bicing in Barcelona. Bicing users may take a bicycle, anchored in one of the many stations through-out the city, and return it to another station after the user reaches his destination. As usual, Bicing records a log of the different tasks that are performed within a process execution. Then, given the following log below:

RegisterNewUser, *PickUpBicycle*, *PickUpBicycle*, *PickUpBicycle*, *PickUpBicycle*

a traditional conformance checking approach could detect it as a conformant, as the model - without considering the data - allows it. However, it may happen that there is a data constraint stating that a user may never have more than three bicycles simultaneously. Hence, the previous trace does not conform to the model, and the results obtained by traditional conformance checking are not accurate enough in this case.

The main contribution of our work is an approach to perform conformance checking on an artifact-centric framework, called BAUML [17,11], which considers not only the flow of the tasks, but also data constraints such as the one illustrated above. Unlike other works, our approach uses a starting framework based on a combination of UML and OCL models (both ISO standard languages, and well-known notations) which covers all the dimensions that should be present in artifact-centric process model [22]. BAUML has been shown to be useful as a specification technique that allows providing automated reasoning over the models [17] while identifying verifiable conditions about them [11].

To briefly sum BAUML up: data or artifacts are represented in the UML class diagram; lifecycles of artifacts are modeled using UML state machine diagram; each external event in the state machine diagram is further defined using a UML activity diagram; and details of the tasks or activities in the activity diagram are represented using OCL operation contracts.

In terms of data conditions, our approach for conformance checking considers the following:

1. Cardinality constraints that establish a maximum number of artifacts (i.e. data objects) to which the current artifact type can be related. For example: a *User* cannot have more than three *Bicycles* simultaneously.
2. Conditions over the transitions in the lifecycles, stating the number of other artifacts to which the current artifact is related. We can deal with two different types:
   (a) When the number of other artifacts is equal to or greater than 0. For example: a *User* can only be unblocked (after being blacklisted) when he no longer has any *Bicycles*.
   (b) When the number of other artifacts is equal to or greater than 1. For example, if an *Active* user has only one bicycle, after returning it he changes his state to *Idle*. However, in this case the transition must decrease the number of related artifacts when it executes. In our previous example, this would mean that the transition decreases the number of bicycles to which the user is related.

We will refer to these data conditions as *context-dependent constraints*, and to the behaviors that depend on them as *context-dependent behavior*.

Bearing this in mind, our problem can be stated as follows. Given: 1. a lifecycle model of an artifact (i.e. a data object), an activity diagram for each external transition in the lifecycle, a set of context-dependent constraints; 2. an event log over the tasks in the activity diagrams where each task is associated to an identifier (i.e. case ID) of the artifact on which the activity is performed; our approach finds deviations between the log and model. Note that we assume that the system registers in the log the IDs of the artifacts involved in each task or event. We check conformance for each artifact individually, and we assume that the event logs have been extracted for each.

We achieve conformance checking by translating the original BAUML model into a Petri Net, while preserving *context-dependent* constraints; and then applying existing conformance-checking techniques to carry out the evaluation between a given event log and the generated Petri Net.

In terms of the contextualization of our contributions within the related work, there are some approaches [25,27,20,18,19] for conformance checking that do consider the data. Both [25,27] incorporate the data and resource perspectives into conformance checking; however, data are merely a set of variables with values.

On the other hand, the following works [20,18,19] consider artifacts, like we do. By using proclets (similar to Petri nets, but including communication ports) as an initial model, they are able to detect deviations when the interaction between artifacts is not correct. This is different from our approach, where we are able to consider certain data-related conditions, such as the maximum number of artifacts that may be related to another artifact, as illustrated before.

As an additional contribution of this paper, we have implemented a plug-in as part of the ProM framework to automatically generate the Petri Net from the original BAUML model; and applied it to perform conformance checking to an example based on real life. In this way, we show the feasibility of our approach in a practical situation.

Finally, we would like to remark that using an artifact-centric model such as BAUML for specifying business processes is also a contribution of our approach since it allows us to bridge the gap between models with a high level of abstraction, which are usually more intuitive and understandable for business people, and notations which are very formal (such as Petri nets) but impractical from the point of view of the business.

The remainder of the paper is structured as follows. Section 2 presents the required preliminaries and Section 3 gives a general overview of our approach. Section 4 formalizes the first step of our approach: translating BAUML models to Petri nets, considering certain data conditions. Section 5 shows how to apply conformance checking, how to interpret the results and how they can be applied back to the original BAUML models. In Section 6 we explain how the approach has been implemented and we evaluate the tool through some tests. Afterwards we examine the related work in Section 7. The last section points out our conclusions and outlines further work.

2 Preliminaries

This section presents the BALSA framework for modeling business processes following an artifact-centric approach and it gives an overview of the basic concepts related to conformance checking and Petri nets, which are necessary to understand this work.

2.1 The BALSA Framework

The BALSA framework [22] establishes four different dimensions that should be present in any artifact-centric business process model:

- **Business Artifacts:** Business artifacts represent the data required by the business. For instance, they may be represented by using an ER model or a UML class diagram. Both diagrams are able to represent the business artifacts, their relationships with other artifacts and to establish constraints on both.
- **Lifecycles:** Lifecycles are used to represent the evolution of an artifact during its life, from the moment it is created until it is destroyed or archived. They can be represented by means of statecharts or condition-action rules.
- **Services:** Services (also known as tasks) are atomic units of work in the business process. As such, they make changes to artifacts by creating, updating and deleting them. They may be represented in different ways: alternatives range from using natural language to logic or through operation contracts.
- **Associations:** Associations establish the execution flow for services. They may be represented using a procedural representation, such as a BPMN or an activity diagram, or using a declarative representation, such as condition-action rules.

2.2 Conformance Checking

In this section we present the necessary notation to represent the event logs, and the Petri nets, the process modeling notation used in most conformance checking approaches.

2.2.1 Event Logs

An execution of a process (also known as *process instance* or *case*) is represented as a sequence of activities called *trace*. An event log is a multiset of traces, representing the behavior observed in the information system during the execution of the process.

Definition 1 (Trace, Event Log) Let $A \subseteq \mathcal{U}_A$ be a set of activities A over an universe of activities $\mathcal{U}_A$. A trace $\sigma \in A^*$ is a sequence of activities. An event log $L \in \mathcal{B}(A^*)$ is a multiset of traces, where $\mathcal{B}(X)$ represents the set of all multisets over X.

2.2.2 Petri Nets

The Petri net notation is a widely extended process modeling notation, and the most used notation in conformance checking given their formal semantics. Petri nets are able to capture concurrency in a natural way, among the other common workflow constructions such as sequence and choice. In this paper, the most common Petri net class is extended to include both more than one token in a place, and the possibility of *inhibitor* arcs (special arcs able to express the empty condition)[1]. The authors assume the reader is familiar with the concept and semantics of Petri nets. For a deeper introduction to Petri nets the reader is referred to [31].

Definition 2 (Petri Net) A *Petri net* over an alphabet A is a tuple $PN = (P, T, F, I, \alpha, M_{ini})$ where P and T represent finite and disjoint set of places and transitions, respectively, and $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation. The initial marking $M_{ini} \subseteq P$ defines the initial state of the system. $I : T \to \mathcal{B}(P)$ indicates the set of inhibitor arcs. Function $\alpha : T \to A \cup \{\tau\}$ maps transitions to activities in A. Transitions mapped to the special symbol τ are considered invisible (graphically represented in color black). The sets of input and

[1] To allow for a simple formalization, these extensions will not be formalized here and only be considered informally when needed in the next section.

output nodes (transitions or places) of a node x, are denoted by $^\bullet x$ and $x^\bullet$, respectively. Finally, since in this work we need to decorate arcs in the Petri net with additional information, we assume that arcs in the Petri net can be annotated over a set of tags $\mathcal{C}$, i.e., there is a function $\eta : F \to \mathcal{C}$.

3 Overview of Our Approach

We aim at defining an approach for applying conformance checking to artifact-centric business process models specified in BAUML, which considers not only the flow of the tasks but also context-dependent constraints. This is achieved by translating the original BAUML model into a Petri Net, in a way that incorporates knowledge from the context-dependent constraints; and then applying existing conformance-checking techniques to carry out the evaluation between a given event log and the generated Petri Net.

The overview of these stages, as seen from the point of view of the tool we have implemented for this purpose, is summarized in Figure 1.

In our environment, users define BAUML models in Visual Paradigm so that we can automatically obtain an XML file with its specification. Other tools could be used as well, provided that they are able to generate the required XML. The BAUML model is then implemented into a system (how this is done is outside the scope of this paper), which should be able to generate the appropriate logs for conformance checking.

Then, at execution time, when processes have been deployed and the system has already generated event logs, we use the tools in the *ProM* framework[2] to perform conformance checking on the BAUML models. This framework includes several plug-ins (not shown in the figure) that allow translating the XML file encoding the BAUML model into a Petri Net and then performing conformance checking from the given event log and the generated Petri Net. These results can then be applied back to the original BAUML specification to update the model or make changes to the deployed system.

Bearing this in mind, the next section (Section 4) formalizes BAUML models and presents how they can be transformed into a Petri net. After this, in Section 5, we explain how to apply conformance checking to the resulting Petri nets and how these results can be transferred to the original BAUML model.

[2] `http://www.promtools.org/`

4 Transforming BAUML models to Petri nets

This section begins by formalizing the BAUML models. It then presents the transformation of these models to Petri nets in order to apply conformance checking techniques, which will allow us detect deviations between the models and reality.

4.1 BAUML models

In this paper we adopt the *BAUML* modeling approach [11,16], which represents the BALSA dimensions using UML [23] and OCL [33]. In particular, we use: UML class diagrams for business artifacts; UML state transition diagrams for lifecycles; UML activity diagrams for associations, and OCL operation contracts for services. This approach also considers *objects*: data in the system whose potential states - resulting from their evolution - are not relevant from the point of view of the business.

More formally, we define a BAUML model $\mathcal{B}$ as a tuple $\langle \mathcal{M}, \mathcal{S}, \mathcal{P}, \mathcal{T} \rangle$, where:

Class Diagram: $\mathcal{M}$ is a UML class diagram, in which some classes represent (business) artifacts. We denote the artifacts as $\textsc{artifacts}(\mathcal{M})$. Intuitively, an artifact $a \in \textsc{artifacts}(\mathcal{M})$ is a class whose evolution results in relevant states from the point of view of the business. We denote the classes in $\mathcal{M}$ as $\textsc{classes}(\mathcal{M})$, and the associations in $\mathcal{M}$ as $\textsc{associations}(\mathcal{M})$. When convenient, we may refer to them as $\textsc{classes}(\mathcal{B})$ and $\textsc{associations}(\mathcal{B})$.

Figure 2 shows the class diagram for our Bicing example with its correspoding textual constraints in OCL. The business artifacts in this case are *Bicycle* and *User*. They can be distinguished from the rest of objects because they have several subclasses, which will keep the information that is relevant in each of the states through which they evolve. In the case of bicycle, it can be *Available*, *InUse*, *Lost* or *Unusable*.

On the other hand, we distinguish between Users who are *Idle*, *Active* or *Blacklisted*. Blacklisted users will not be allowed to make any new rentals, but may still have bicycles in their possession. Active users have at least one current rental while idle users have none. Note that users may be using up to three bicycles. This allows families to rent bicycles without having to register their children as users.

State Transition Diagrams: $\mathcal{S}$ is a set of UML state transition diagrams, one per artifact in $\textsc{artifacts}(\mathcal{M})$. More formally, for each artifact $\mathtt{A} \in \textsc{artifacts}(\mathcal{M})$, $\mathcal{S}$ contains a state transition diagram $S_{\mathtt{A}} = \langle V, v_o, v_f, E, X, T \rangle$, where V is a set of states, $v_o \in V$

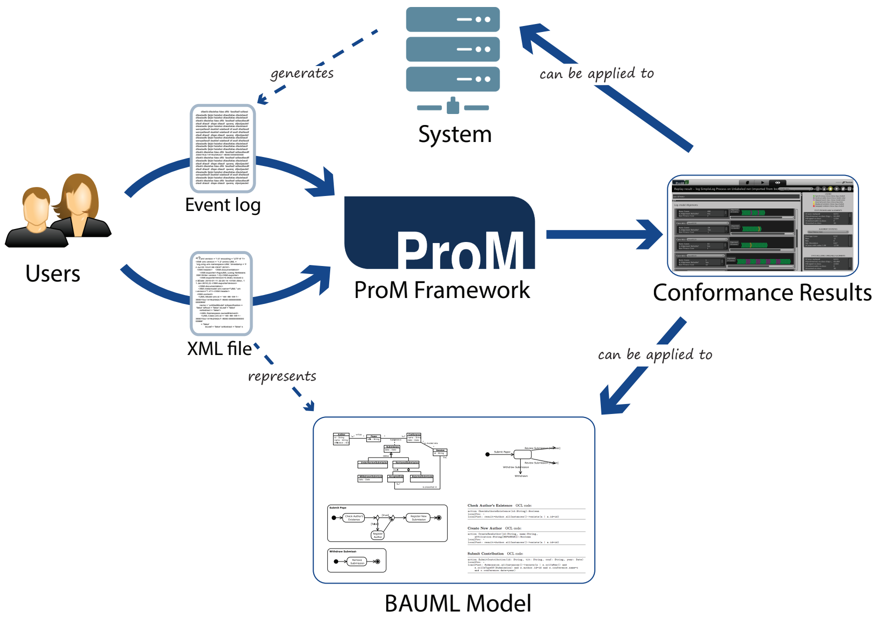
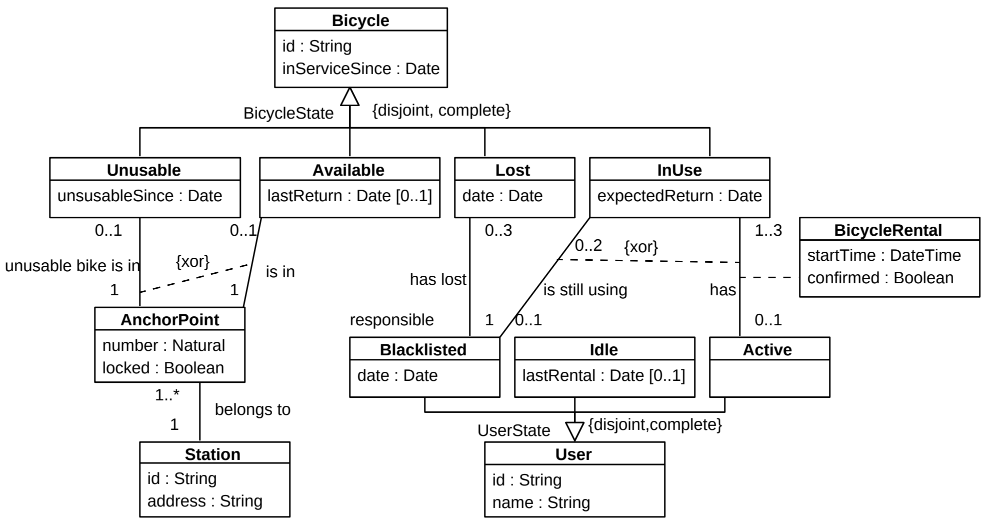


Fig. 1: Overview of the process to apply conformance checking to BAUML models



1. *Station*, *User* and *Bicycle* are identified by their respective *id* (only the OCL for *Station* is shown):

```
context Station inv: self.id->isUnique()
```

2. An *AnchorPoint* is identified by its *number* and the *station* it belongs to:

```
context Station inv: self.anchorPoint.number->isUnique()
```

3. A *Blacklisted* user may not have more than 3 *Bicycles*:

```
context Blacklisted inv: self.lost->size()+self.inUse->size() <= 3
```

Fig. 2: Class diagram showing the artifacts and objects in Bicing.

is the initial state, $v_f \in V$ is the final state, E is a set of events (either *external* or *time* events), X is a set of effects (i.e. an atomic task or service), and $T \subseteq V \times OCL_{\mathcal{M}} \times E \times C \times X \times V$ is a set of transitions between pairs of states, where $OCL_{\mathcal{M}}$ is an OCL condition over $\mathcal{M}$ that must be true in order for the transition to take place and C is a tag on the result of the execution of the event in E. We will denote the external events in $S_{\mathtt{A}}$ as $\textsc{extEvents}(S_{\mathtt{A}})$.

Each external event is further defined in an activity diagram. Each effect X corresponds to an atomic task to be performed when making the transition, and whose parameters are exactly the artifacts involved in the transition.

As artifacts cannot evolve from one state to another randomly, the state machine diagram shows how the transitions from one state to the next are triggered. Figures 3 and 4 show the state machine diagrams that represent the evolution of the artifacts bicycle and user.

When a bicycle is created (*Register New Bicycle*), it is in state *Available*, and ready to be picked up. If a user does pick up a bicycle (event *Pick Up Bicycle*), there are two possibilities: either everything goes smoothly and he takes it with him (tag *success*), or the bicycle is in bad shape and the pick-up fails (tag *fail*). In the first case, the bicycle is *InUse* while in the second case it is *Unusable*, following our assumptions. While a bicycle is in state *InUse*, either the user returns the bicycle (*Return Bicycle*) or he keeps it for more than three days. If the latter, the user must be blacklisted and the bicycle changes its state to *Lost*. If a lost bicycle is recovered (*Recover Bicycle*), then it is *Unusable*. A repair of an unusable bicycle (*Repair Bicycle*) may be either successful (tag *success*), making the bicycle *Available* again or it may fail causing the deletion of the bicycle because it is beyond repair.

The evolution of a user is similar to that of a bicycle, and it shares many of its events. However, notice that in some cases there are conditions over the transitions. For instance, if an *Active* user returns a bicycle (event *Return Bicycle*), the final state will depend on whether there is only one bicycle left to return (*#bicycle=1*) or more (*#bicycle>1*).

This condition *#bicycle* refers to number of bicycles the user has and is defined in OCL as `self.oclAsType(Active).inUse->size()` (for transitions with source state *Available*) or as `self.oclAsType(Blacklisted).inUse -> size() + self.oclAsType(Blacklisted).lost -> size()` (for transitions with source state *Blacklisted*).

Activity Diagrams: $\mathcal{P}$ is a set of UML activity diagrams, such that for every state transition diagram $S{=}\langle V, v_o, v_f, E, X, T\rangle \in \mathcal{S}$, and for every event $\varepsilon \in \textsc{extEvents}(S)$ there exists exactly one activity diagram $P_\varepsilon \in \mathcal{P}$.

P_ε is a tuple $\langle N, n_o, n_f, F\rangle$, where N is a set of nodes, $n_o \in N$ is the initial node, $n_f \subset N$ is the set of final nodes and $F \subseteq N \times G \times C \times N$ is a set of transitions between pairs of nodes where C is a tag (`success` or `fail`) denoting the correct or incorrect execution of the transition, and G a guard condition.

There are four different types of nodes $n \in N$ in an activity diagram P_ε: initial nodes (denoted as $\textsc{ini}(P_\varepsilon)$), final nodes ($\textsc{final}(P_\varepsilon)$), gateways ($\textsc{gateways}(P_\varepsilon)$ and activities ($\textsc{activities}(P_\varepsilon)$). *Gateways* are used to define the control flow. They may be either a *decision node*, a *merge node*, an *inclusive-or node*, a *fork node* or a *join node*.

An *activity* may be an (atomic) *task* or a *material action*. Each task is associated to an *operation contract*, which expresses a precondition on the executability of the task, and a postcondition describing its effect, both formalized in terms of OCL queries over $\mathcal{M}$. *Material actions* represent physical work that is done in the process but that does not change the system.

We make the following assumptions over each activity diagram P_ε: decision nodes and fork nodes have one incoming flow and more than one outgoing flow; merge nodes and join nodes have more than one incoming flow and exactly one outgoing flow; and tasks have exactly one incoming and one outgoing flow.

With a slight abuse of notation, given a state transition diagram $S \in \mathcal{S}$, we denote by $\mathcal{P}_S \subseteq \mathcal{P}$ the set of activity diagrams referring to all external events appearing in S.

Returning to our running example, each external event in the state machine diagrams in Figures 3 and 4 has its corresponding activity diagram. Figure 5 shows the details for transition *Pick Up Bicycle*. In this case, the user requests the bicycle (*RequestBicycle*) and he physically gets it (*Get Bicycle*). If the bicycle is in bad shape (e.g. it is missing a pedal), then the user places it again in its anchor point (*Return to Anchor Point*) and the return is confirmed (*Confirm Return*). Otherwise, he confirms the pick-up. The two tags placed over the final edges at the end of the activity diagram, *succeed* and *fail*, are used to differentiate between a successful pick-up and a failed one.

Tasks: $\mathcal{T}$ is a set of atomic tasks, each of which has an OCL operation contract. Its semantics is that the task can only be executed when the current information base satisfies its precondition, and that, once executed, the task brings the information base to a new state that satisfies its postcondition.

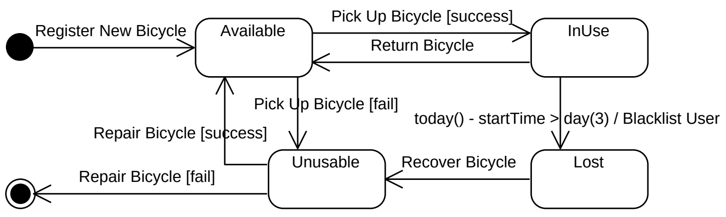


Fig. 3: State machine diagram showing the evolution of artifact Bicycle.

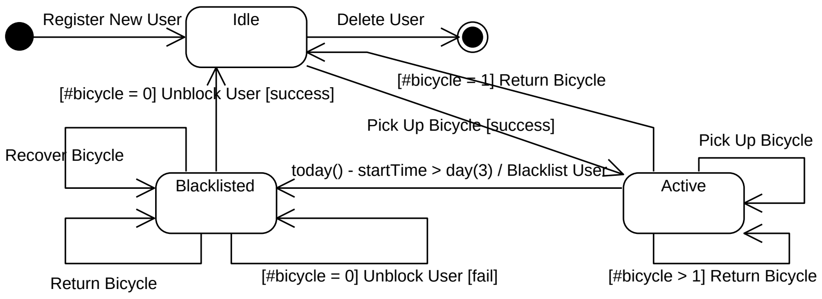

Register New User
Idle
Delete User
[#bicycle = 1] Return Bicycle
[#bicycle = 0] Unblock User [success]
Pick Up Bicycle [success]
Pick Up Bicycle
Recover Bicycle
today() - startTime > day(3) / Blacklist User
Blacklisted
Active
Return Bicycle
[#bicycle = 0] Unblock User [fail]
[#bicycle > 1] Return Bicycle

Fig. 4: State machine diagram showing the evolution of artifact User.

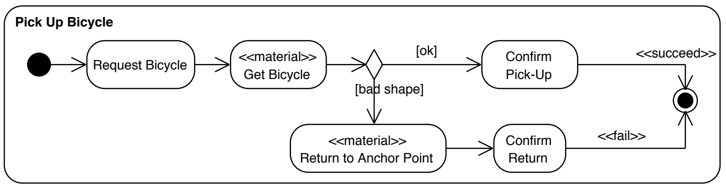


Fig. 5: Activity diagram showing the details of transition *Pick Up Bicycle*.

Given an artifact $\mathtt{A} \in \mathcal{M}$, we denote by $\textsc{tasks}(\mathtt{A})$ the set of tasks appearing in the state transition diagram $S_{\mathtt{A}}$, also considering all activity diagrams related to $S_{\mathtt{A}}$. We then define $\textsc{tasks}(\mathcal{B}) = \bigcup_{\mathtt{A} \in \textsc{artifacts}(\mathcal{M})} \textsc{tasks}(\mathtt{A})$.

For instance, the operation contract for task *Confirm Pick-Up* can be seen below:

```
operation Confirm Pick-Up (u:User, b:Bicycle)
post: BicycleRental.allInstances()->exists(br
    | br.oclIsNew() and br.startTime=now()
    and confirmed=true and
    br.inUse=b.oclAsType(InUse) and
    br.active=u.oclAsType(Active))
```

Confirm Pick-Up has two input parameters, *User* and *Bicycle*. Note that this task is part of an event (*Pick Up Bicycle*) which appears in two different state machine diagrams, for artifacts *User* and *Bicycle*. Hence, it has the two artifacts as input parameters.

For the purpose of this paper, we assume that the whole BAUML is coherent and that the various models conform to each other; e.g. each artifact has its corresponding state machine diagram, the external events are refined in an activity diagram, the tags in the activity diagram appear in the corresponding transitions of the state machine diagram, etc.

4.2 Converting BAUML models to Petri nets

Logs contain information about the events that have taken place in a system, including when they happened. In the context of process conformance, an event is a noteworthy occurrence whose execution is recorded in a log. In the context of BAUML models, an event may actually be a set of occurrences.

In order to check conformance of these logs to our BAUML models, we need to be able to determine which components of BAUML actually deal with occurrences that are recorded in the log. According to our representation, external events deal with noteworthy occurrences. In particular, external events are further refined by means of activity diagrams showing the units of work (i.e. the *events* in the context of process conformance) and the order in which they should occur.

However, the occurrence of these events and effects cannot take place randomly at any time. Their existence is limited by the state of the artifact that they make changes to. This information is represented in the state machine diagram, which shows the artifact's evolution. For this reason, when translating our specification into Petri nets with the final goal of checking conformance, we should translate activity diagrams, which contain the details of external events, and the context in which events occur, defined by the state machine diagram.

This section will present this translation process. The translation in essence is similar to the one done in [28] for BPMN extended with data annotations.

At the beginning, given a BAUML model $\mathcal{B} = \langle \mathcal{M}, \mathcal{S}, \mathcal{P}, \mathcal{T} \rangle$, we have a set $\textsc{artifacts}(\mathcal{M})$ of business artifacts. For each $\mathtt{A} \in \textsc{artifacts}(\mathcal{M})$, $S_\mathtt{A}$ corresponds to its state machine diagram and $\mathcal{P}_{S_\mathtt{A}}$ corresponds to the set of activity diagrams for each external event ε in $(S_\mathtt{A})$. Given an artifact $\mathtt{A}$, we proceed in two steps. First, the transformation of $S_\mathtt{A}$ into a Petri net $\overline{\mathcal{PN}}_\mathtt{A}$ is performed. In the second step, each transition t in $\overline{\mathcal{PN}}_\mathtt{A}$ that contains an external event ε is refined with the Petri net $PN(P_\varepsilon)$ representing the corresponding behavior underlying activity diagram P_ε. After refining these transitions in $\overline{\mathcal{PN}}_\mathtt{A}$, the final Petri net $\mathcal{PN}_\mathtt{A}$ representing artifact $\mathtt{A}$ will be obtained.

4.2.1 From S_A to $\overline{\mathcal{PN}}_A$

The translation process from $S_\mathtt{A}$ to $\overline{\mathcal{PN}}_\mathtt{A}$ will map each component of the state machine diagram $S_\mathtt{A}$ to one or several components of the resulting Petri net $\overline{\mathcal{PN}}_\mathtt{A}$.

To begin with, each state $v \in V$ in $S_\mathtt{A} = \langle V, v_o, v_f, E, X, T \rangle$ will correspond to a place $pl \in Pl$ in $\overline{\mathcal{PN}}_\mathtt{A} = (Pl, Tr, Ar, \emptyset, \alpha, M_{ini})$. The place $pl_o \in Pl$ derived from the initial state v_o of $S_\mathtt{A}$ will be marked with one token such that $M_{ini} = \{pl_o\}$.

State transitions in $S_\mathtt{A}$ are a bit more complex to translate. Intuitively, and in the general case (i.e. the transition has no tags associated to its event), given a transition t in $S_\mathtt{A}$, which has v_s and v_t as source and target states (respectively), the transformation is straightforward: a transition tr and two arcs (pl_s, tr) and (tr, pl_t) will be added to the petri net $\overline{\mathcal{PN}}_\mathtt{A}$, where pl_s and pl_t are the places that correspond to v_s and v_t.

More formally, transitions in T will be translated according to their type as explained below (see section section 2.1 for their definition). Note that at this point we do not consider yet conditions $OCL_\mathcal{M}$:

1. `TimeEvent /`X
2. `ExternalEvent(`$a_1, ..., a_n$`) ([`C`])`

We define two subsets of transitions, $T_{time} \subseteq T$ and $T_{ext} \subseteq T$ that correspond to the first and second type of transitions respectively.

Each transition in $S_\mathtt{A}$ of the first type will correspond to a transition in the Petri net, labelled with the name of effect x. Time events are not specifically translated. More formally, for each $t \in T_{time}$ such that $t = \langle v_i, \emptyset, e, \emptyset, x, v_t \rangle$, we define a transition tr in the Petri net labelled x. We add two arcs $(pl_i, tr), (tr, pl_t)$ to the arc set Ar, where pl_i and pl_t correspond to the places of v_i and v_t.

For the second type of transitions, which denote external events, we first add a transition tr which will represent a place-holder for all the possible outcomes of the external event, thus hiding the choices. Afterwards, the actual choices will be refined. We first need to group in the same set those transitions in $S_\mathtt{A}$ which have the same source state and event but a different tag. The remaining transitions (i.e. the ones which are not grouped into a set) will be dealt with in a different way. The reason behind this is that each set of transitions in $S_\mathtt{A}$ will map to only one transition in the Petri net $\overline{\mathcal{PN}}_\mathtt{A}$, whereas the remaining ones will each correspond to a transition.

More formally, for every set of transitions $T_{cond_m} = \{t_1, ..., t_k\} \subseteq T_{ext}$ such that $t_1 = \langle v_s, \emptyset, e, c_1, \emptyset, v_{t_1} \rangle$, ..., $t_k = \langle v_s, \emptyset, e, c_k, \emptyset, v_{t_k} \rangle$, where for each i, j such that $i \neq j$ and $1 \leq i, j \leq k$, then $c_i \neq c_j$, and $e \in \textsc{extEvents}(S_\mathtt{A})$ the whole set of transitions will correspond to one transition $tr \in Tr$ in $\overline{\mathcal{PN}}_\mathtt{A}$ labelled e. In this case, $k+1$ arcs are added to Ar: the arc (pl_s, tr) and k different arcs $(tr, pl_1)[c_1], \ldots, (tr, pl_k)[c_k]$. Hence, these k arcs are annotated with their corresponding tag, and each pl_i corresponds to the place representing state s_i[3].

Figure 6 shows how the Bicycle artifact is translated in this first step. Notice for instance transitions *Pick Up Bicycle [success]* and *Pick Up Bicycle [fail]* in Figure 3. Both transitions have the same state as source and the same event, so the target state in this case is determined by the tag associated to the transition. In the derived Petri net (Figure 6), the two possibilities are represented as two arcs from the same transition *PickUp*.

Strictly speaking, the Petri net $\overline{\mathcal{PN}}_\mathtt{A}$ that results from this step is not equivalent to the original $S_\mathtt{A}$. The reason is the way in which we translate transitions which contain events with tags. An event e in $S_\mathtt{A}$ leads to a state s_k only if the tag c_k is satisfied, whereas if the traditional Petri net firing rule is considered in $\overline{\mathcal{PN}}_\mathtt{A}$ (i.e., leaving out the conditions annotated in the outgoing arcs of e in $\overline{\mathcal{PN}}_\mathtt{A}$), the place connected via the arc annotated with c_k will always be marked. Again, we stress that the transformation in this first step is only used to facilitate the incorporation of the Petri nets that correspond to the activity diagrams into

[3] We overload the syntax of the Petri nets to incorporate in some of the arcs information on the tag used. This is only to facilitate the translation in the next step, and has no semantics associated.

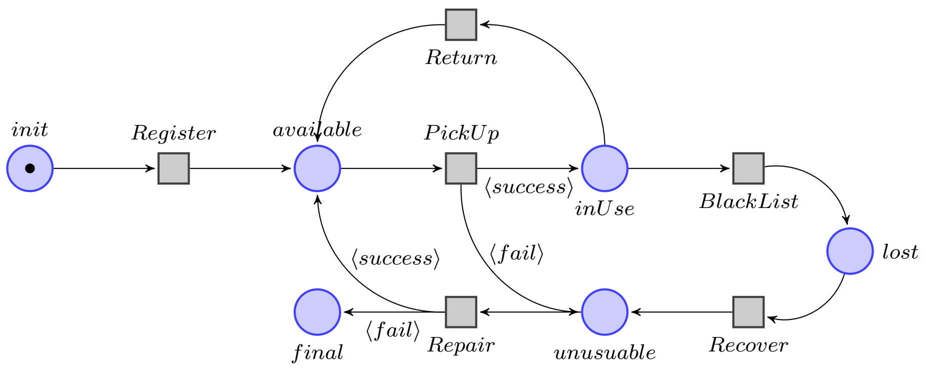


Fig. 6: The *bicing* example, representing the artifact shown in Figure 3.

this net, and therefore the problem will be amended at the end. In the Bicing example, we will see how the supposed concurrency between places *inUse* and *unusable* in the Petri net of Figure 6 will never happen after the transformation done in the next step.

4.2.2 From $\overline{\mathcal{PN}}_{\mathtt{A}}$, $\mathcal{P}_{S_{\mathtt{A}}}$ to $\mathcal{PN}_{\mathtt{A}}$

After the first step has been performed, we need to redefine some of the transitions in $\overline{\mathcal{PN}}_{\mathtt{A}}$. In particular, we will redefine those that correspond to an external event in $S_{\mathtt{A}}$ and, for this reason, they have an associated activity diagram.

Each such transition $tr_\varepsilon \in Tr$ in $\overline{\mathcal{PN}}_{\mathtt{A}}$ labelled with event ε will be refined with the Petri net obtained by translating the corresponding activity diagram P_ε, denoted $\mathcal{PN}(P_\varepsilon)$. For the translation of activity diagrams into Petri nets, we will base our translation on the work of [38]. We will restrict the translation to those elements that are relevant for the specification of associations within the BAUML framework. These elements are actions (i.e. tasks), decision, merge, fork and join nodes. We will also deal, logically, with the initial node and the activity final node. Figure 7, based on [38], shows how to translate each of these elements (left column) to an element in a Petri net (right column).

Also notice that transition *tr* will only have one outgoing and one incoming arc. There can only be one outgoing arc because after its execution the artifact will be in a specific state and not in several states at once. There is only one incoming arc because a transition in this case corresponds to an arc in $S_{\mathtt{A}}$, which can only have one source state. Applying this translation to our activity diagram in Figure 5, we obtain the Petri net in Figure 8. Note that guard conditions are not specifically translated.

Without loss of generality, we assume that the elements in $\mathcal{PN}(P_\varepsilon)$ and $\overline{\mathcal{PN}}_{\mathtt{A}}$ are disjoint. In order to determine how we should refine each transition $tr_\varepsilon \in Tr$ in $\overline{\mathcal{PN}}_{\mathtt{A}}$ labelled with event ε such that $\varepsilon \in$ EXTEVENTS($S_{\mathtt{A}}$) with $\mathcal{PN}(P_\varepsilon) = (Pl_\varepsilon, Tr_\varepsilon, Ar_\varepsilon, I_\varepsilon, \alpha, M_{ini,\varepsilon})$, we will consider two different cases:

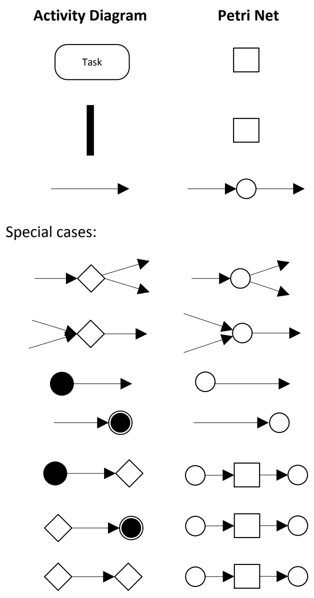


Fig. 7: Translation of activity diagram's elements into a Petri net, adapted from [38].

1. P_ε does not have any tags in the edges connected to the activity final node. In terms of the semantics of the model, this means that the execution of P_ε will bring the artifact to the state indicated in the transition system without any conditions. Formally, given $P_\varepsilon = \langle N, n_o, n_f, F\rangle$, every $f = \langle n, g, c, n_f\rangle \in F$ (note that n_f is a final node) implies that $c = \emptyset$.
2. P_ε does have tags in the edges connected to the final node. This means that the artifact's final state after the execution of P_ε depends precisely on the result of this execution. In this case, there will be tags

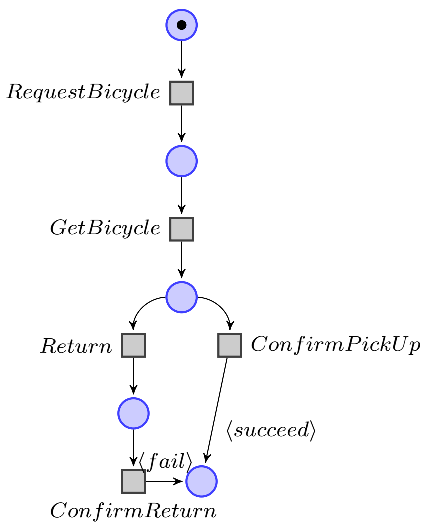


Fig. 8: Petri net resulting from the translation of activity diagram Pick Up Bicycle in Figure 5.

associated to the outgoing arcs from tr_ε in $\overline{\mathcal{PN}}_{\mathtt{A}}$. Formally, given $P_\varepsilon = \langle N, n_o, n_f, F\rangle$, at least one $f = \langle n, g, c, n_f\rangle$ (note that n_f is a final node) implies that $c \neq \emptyset$.

In the first case, tr_ε will be substituted by $\mathcal{PN}(P_\varepsilon)$ by adding all the nodes and arcs of $\mathcal{PN}(P_\varepsilon)$ except the initial and final place (and corresponding arcs). For these, the place $pl \in {}^\bullet tr_\varepsilon$ will be connected to the outgoing arcs of the initial place M_{ini}: let $(M_{ini}, z_1), \ldots, (M_{ini}, z_k)$ be a set of arcs in $\mathcal{PN}(P_\varepsilon)$; the set of arcs $(pl, z_1), \ldots, (pl, z_k)$ is inserted into $\overline{\mathcal{PN}}_{\mathtt{A}}$. The transformation is symmetric for the place $pl' \in tr_\varepsilon^\bullet$ and the final place of $\mathcal{PN}(P_\varepsilon)$.

In the second case, tr_ε will be substituted by $\mathcal{PN}(P_\varepsilon)$ by adding all the nodes and arcs of $\mathcal{PN}(P_\varepsilon)$ except the initial place and the final places (and corresponding arcs). Like in the previous case, the place $pl \in {}^\bullet tr_\varepsilon$ will inherit the outgoing arcs of the initial place M_{ini}: let $(M_{ini}, z_1), \ldots, (M_{ini}, z_k)$ be the set of arcs in $\mathcal{PN}(P_\varepsilon)$; the set of arcs $(pl, z_1), \ldots, (pl, z_k)$ is inserted into $\overline{\mathcal{PN}}_{\mathtt{A}}$.

For every final place $pl_f \in \mathcal{PN}(P_\varepsilon)$, given the arcs $(tr_1, pl_f)[c_1], \ldots, (tr_n, pl_f)[c_k]$, every place $pl' \in tr_\varepsilon^\bullet$ in $\overline{\mathcal{PN}}_{\mathtt{A}}$ will have them as incoming arcs of the final place pl_f. In this way we avoid the supposed concurrency between places in the Petri net $\overline{\mathcal{PN}}_{\mathtt{A}}$ obtained as a result of the first step.

Figure 9 describes the Petri net obtained from the complete BAUML description of the artifact *Bicycle* of our example in Section 4.1.

4.3 Adding context-dependent behaviour to Petri nets

In the previous section we have shown how to obtain a Petri net for a state machine diagram and its corresponding activity diagrams, but we did not consider any conditions $OCL_{\mathcal{M}}$ in the state machine diagram.

However, in general, the execution of a business artifact may depend on the execution of other business artifacts as it happens in our example with artifact *User* which depends on the behavior of *Bicycle*. This kind of situations can be easily identified from the artifact-centric specification since the state machine diagram of those instances of artifacts will contain some transition conditioned on the number of other instances. The state diagram of *User* in Figure 4 contains several such transitions that depend on the number of bicycles that the user still has. In this case, the translation from the state machine diagram into a Petri Net is an extension of the previous one.

States and transitions in state machine $S_{\mathtt{A}} = \langle V, v_o, v_f, E, X, T\rangle$ are encoded in a Petri net $\overline{\mathcal{PN}}_{\mathtt{A}} = (Pl, Tr, Ar, \emptyset, \alpha, M_{ini})$ as explained in Step 1. So, we need to explain only the different kinds of transitions whose behaviour is related to the number of other artifacts. Let $tr \in Tr$ be a transition in $\overline{\mathcal{PN}}_{\mathtt{A}}$, such that $pl \in {}^\bullet tr$ and $pl' \in tr^\bullet$ and tr is labelled with event $e[t]$ or e, or effect x (but not both); the following aspects need to be taken into consideration:

1. The number (i.e. the cardinality) of artifacts to which the current artifact is related.
2. Whether the event e or the effect x in the transition tr in $S_{\mathtt{A}}$ changes the number of other artifacts related to the current one.
3. Whether the triggering of the transition tr depends on a guard condition that checks the number of other artifacts related to the current instance of the artifact. This is similar to the work in [18].

The remainder of this subsection is structured according to these three points.

4.3.1 Incorporating cardinalities into the Petri net

As Figure 10 shows, for each other artifact B referenced in $S_{\mathtt{A}}$, we need to incorporate two different counters (complementary k-bounded places in the Petri net) in $\overline{\mathcal{PN}}_{\mathtt{A}}$ for that purpose: $available_B$, with as many tokens as other artifacts can be taken; and $taken_B$, with denotes the tokens currently taken (initially it has no token).

The number of tokens to be added to the place $available_B$ corresponds to the maximum cardinality between artifacts A and B; more specifically, to the maximum number of B that any point in time can be related

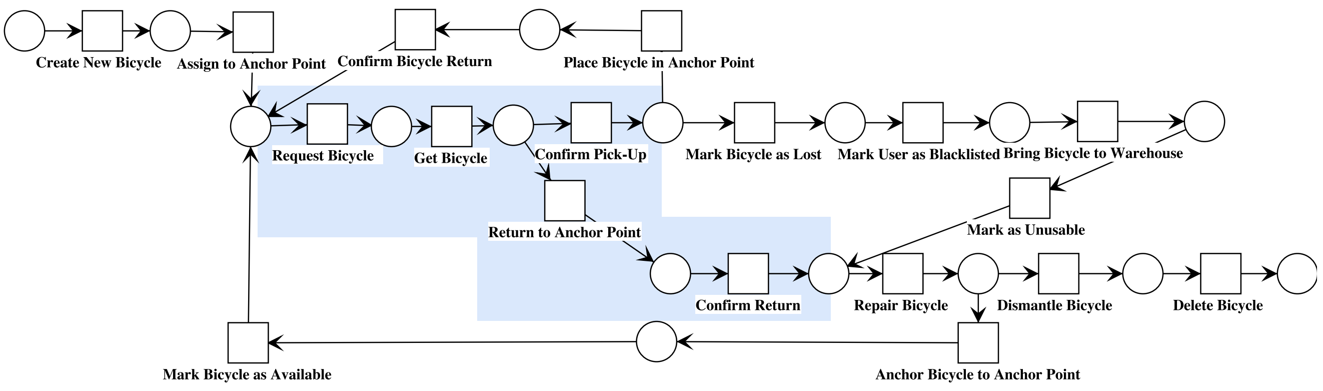


Fig. 9: Petri net PN_{bike} describing the Bicycle artifact, highlighting the *Pick up Bicycle* activity diagram translation part.

to A. This information can be extracted from the class diagram, see Figure 2.

Intuitively, this can be done by examining all the associations between the artifacts (both at the superclass and the subclass level), and keeping the highest maximum cardinality.

This process is detailed in Algorithm 1. Given two artifacts, A and B, the algorithm begins by examining the superclass of A and finding out the highest maximum cardinality of B in relation to A's superclass. This value is stored in a variable. Afterwards, the algorithm applies the same process to each subclass of A, this time only keeping the highest maximum cardinality considering all the subclasses (remember that due to the *disjoint, complete* constraint in the artifact's hierarchy an instance of an artifact can only have one of the subclasses types at a time). Finally, these two values are added to obtain the highest maximum cardinality, that is, the number of B artifacts that can be related at most to A.

Algorithm 1 calculateHighestMaxCardinality(A,B)

```
$a_{sup}$ is artifact's $A$ top superclass
$A'$ is the set of $A$'s subclasses
int $highCardSup = highMaxCardForClass(a_{sup}, B)$;
int $highCardSub = 0$;
for all $a \in A'$ do
    int $currHighCardSub = highMaxCardForClass(a, B)$;
    if $highCardSub < currHighCardSub$ then
        $highCardSub = currHighCardSub$;
    end if
end for
return $highCardSup + highCardSub$
```

Algorithm 2 details how the highest maximum cardinality between a certain class a and an artifact B is obtained. At the start it obtains all the relationships between a and any class in B's hierarchy. It then generates all valid maximum combinations of relationships. If there are no XOR constraints between a and any class of B, then there will only be one maxium combination, containing all the relationships between a and any B. On the other hand, if there any XOR constraints, there will be at least two valid maximum combinations that need to be considered.

In our example, artifact *User* has no relationships with *Bicycle* at the superclass level. On the other hand, a *Blacklisted* user may be related to *Lost* and *InUse* bicycles. Since there is no XOR constraint between these two relationships, there will be only one maximum valid combination, formed by these two relationships.

For each of these valid combinations, it calculates their maximum cardinality and keeps only the highest.

Algorithm 2 highMaxCardForClass(a,B)

```
$R$ is the set of associations between $a$ and any $b \in B$
Set⟨ Set⟨ Relationship⟩⟩ $relComb = getValidComb(R)$;
int $globalHighCard = 0$;
for all $relSet \in relComb$ do
    int $highCard = calculateHighCard(relSet)$;
    if $highCard > globalHighCard$ then
        $globalHighCard = highCard$;
    end if
end for
return $globalHighCard$;
```

Finally, Algorithm 3 shows how the highest maximum cardinality for each of these combinations is obtained. If there is a relevant OCL constraint of the form `context A inv: self.roleB1->size() + ... + self.roleBn->size() <= Y` , the algorithm will obtain the maximum cardinality established by the constraint (Y) and consider all the classes that take part in it as already processed.

On the other hand, if there aren't relevant OCL constraints as described above, then the highest maximum cardinality will be obtained by adding up all the maximum cardinalities at the B end of the relationships.

Note that we impose a restriction on OCL constraints of the form `context A inv: self.roleB1->size() + ... + self.roleBn->size() <= Y` , by stating that a certain relationship (identified by `self.roleBi`) may not appear more than once in such constraints with the same (`context A`).

Algorithm 3 calculateHighCard(relSet)

```
int $highestCard = 0$;
for all $r \in relSet$ do
    if $r$ appears in relevant OCL constraint then
        int $currentMaxCard = getMaxCardOclConstr(r)$;
        Set⟨ Relationship⟩ $relOCL = getRelInOclConstr(r)$;
        $relSet.removeSet(relOCL)$;
        $highestCard += currentMaxCard$;
    else
        int $currentMaxCard = getMaxCard(r)$;
        $highestCard += currentMaxCard$;
    end if
end for
return $highestCard$;
```

Returning to our example, the relationships between *Blacklisted* user, and *InUse* and *Lost* bicycle do not have any XOR constraint. However, there is an OCL constraint that follows the pattern explained above (`context Blacklisted inv: self.lost->size()+self.inUse->size() <= 3`). In this case, Algorithm 3 will determine that the maximum cardinality is 3, as stated in the constraint.

4.3.2 Incorporating cardinality updates to the Petri net

For every transition transition *tr* in $S_{\mathtt{A}}$ we need to do the following: 1) determine if *tr* increases or decreases the number of other artifacts (or does neither), 2) in case it makes changes, incorporate them into the Petri net.

Determining changes to the number of artifacts. If the event *e* or the effect *x* in the transition *tr* in $S_{\mathtt{A}}$ changes the number of other artifacts related to the current one, two situations need to be considered:

(a) *tr* increases the number of other artifact instances.
(b) *tr* decreases the number of other artifact instances.

The knowledge required to determine whether an event modifies the number of other artifacts can be automatically drawn from the operation contracts specifying the services in the activity diagrams. We do so by identifying the structural events in the postcondition (i.e. elementary changes in the population of an entity or relationship type [32]). In particular we need to look for structural events that add or delete instances of the relationship between the artifacts (including their subclasses).

Below, we present some OCL patterns [35] that result in the structural events that create or delete instances of the relationship between the artifacts. This also includes the creation and deletion of association classes, whose existence is tied to the existence of the relationship between the classes:

– **Relationship Creation:** An instance of a relation between classes $\mathtt{c}_i$ and $\mathtt{c}_j$ will be created by any expression of the form $\mathtt{c}_i$`.role-`$\mathtt{c}_j$ `=` $\mathtt{c}_j$ or $\mathtt{c}_i$`.role-`$\mathtt{c}_j$`-> includes(`$\mathtt{c}_j$`)`, where $\mathtt{c}_i$ and $\mathtt{c}_j$ refer to classes in the class diagram, while `role-`$\mathtt{c}_j$ refers to the relationship between $\mathtt{c}_i$ and $\mathtt{c}_j$ through `role-`$\mathtt{c}_j$.
– **Relationship Deletion:** An instance of a relation between $\mathtt{c}_i$ and $\mathtt{c}_j$ will be deleted if it contains the OCL expression $\mathtt{c}_i$`.role-`$\mathtt{c}_j$ `->excludes(`$\mathtt{c}_j$`)`, or if an instance `x` of $\mathtt{C}_i$ or $\mathtt{C}_j$ is deleted. The deletion of an instance `x` of a class `C` will contain the expression $\mathtt{C}_{gen}$`.allInstances()->excludes(x)` or the expression: `not x.oclIsTypeOf(`$\mathtt{C}_{gen}$`)`, where $\mathtt{C}_{gen}$ is either the class `C` or a superclass of `C`.
– **Association Class Creation:** In this case, the creation would have the form: `C.allInstances()-> exists(i| i.oclIsNew() and i.part`$_1$`=p`$_1$ `and...and i.part`$_n$`=p`$_n$ `and i.attr`$_1$`=a`$_1$ `and...and i.attr`$_m$`=a`$_m$`)` or the expression: `i.oclIsTypeOf(C) and i.part`$_1$`=p`$_1$ `and ... and i.part`$_n$`=p`$_n$ `and i.attr`$_1$`=a`$_1$ `and...and i.attr`$_m$`=a`$_m$, where $\mathtt{p}_i$ refers to any class $\mathtt{C}_i$ participating in the relationship, and $\mathtt{attr}_j$ to any attribute of `C`.
– **Association Class Deletion:** The deletion of an association class `i` has the form `C.allInstances()-> excludes(i)`, or: `not i.oclIsTypeOf(C)`. It will also be deleted if any of its participants $\mathtt{p}_1,\ldots,\mathtt{p}_n$ is deleted.

For example, for the OCL operation contract of task *Confirm Pick-Up* (see page 7), we can see that a new *BicycleRental* is created (an association class), linking a *Bicycle* and a *User*. Hence, this increases by one the number of bicycles assigned to the user.

Incorporating these changes to the Petri net. We now need to include additional arcs/inhibitor arcs in $\overline{\mathcal{PN}}_{\mathtt{A}}$ to be able to model the effect of the events on the counters, and to ensure for instance that a transition may only take place when there is not an instance of an artifact taken. In the following transformations, each time a new place or transition is created we assume that new names are created for these elements, so that name clashes with previouly created elements do not arise. This is done as follows:

(a) If tr increases the number of instances of B, a new place pl_B is added to $\overline{\mathcal{PN}}_{\mathtt{A}}$ and the arc (tr, pl') is replaced by arc (tr, pl_B). A new invisible transition $incCounter_B$ is added to $\overline{\mathcal{PN}}_{\mathtt{A}}$ together with arcs: $(pl_B, incCounter_B)$, $(available_B, incCounter_B)$, $(incCounter_B, taken_B)$ and $(incCounter_B, pl')$.
(b) If tr decreases the number of instances of B, a place pl_B is added to $\overline{\mathcal{PN}}_{\mathtt{A}}$ and the arc (tr, pl') is replaced by arc (tr, pl_B). A new invisible transition $decCounter_B$ is added to $\overline{\mathcal{PN}}_{\mathtt{A}}$ together with arcs: $(pl'', decCounter_B)$,$(taken_B, decCounter_B)$, $(decCounter_B, available_B)$ and $(decCounter_B, pl')$.

Note that events/tasks that merely change the subclass of an artifact (by deleting the initial subclass and adding a new different subclass, but keeping the relationships between artifacts), do not require additional arcs or inhibitor arcs, as they keep the balance between available and taken tokens. This results in a simplified, more streamlined Petri net.

4.3.3 Incorporating guard conditions to the Petri net

The last step incorporates the guard conditions into the Petri net. More specifically, if the triggering of the transition tr depends on a condition that checks the number of other artifacts related to the current instance of the artifact, we consider two alternatives:

(a) Equality or inequality comparison to 0.
(b) Equality or inequality comparison to 1. In this case, the event or effect of the transition is assumed to be decreasing the number of instances that are referred to by the transition[4].

Note that guard conditions referring to the number of other artifacts related to the current one is represented with $\#ArtifactName$. In our example, it is $\#bicycle$.

(a) If tr contains an inequality or equality comparison to 0 on the number of instances of B:
  (i) If it is an inequality comparison, a new place pl_B is added to $\overline{\mathcal{PN}}_{\mathtt{A}}$ and the arc (pl, tr) in $\overline{\mathcal{PN}}_{\mathtt{A}}$ is replaced by arc (pl_B, tr). A new invisible transition $continue_B$ is added to $\overline{\mathcal{PN}}_{\mathtt{A}}$ together with arcs: $(pl, continue_B)$, $(taken_B, continue_B)$, $(continue_B, taken_B)$ and $(continue_B, pl'')$.
  (ii) Using the same place pl_B, we proceed similarly in the case of an equality comparison, with a new invisible transition $back_B$, with arcs $(pl, back_B)$ and $(back, pl_B)$ and a zero comparison from $taken_B$ to $back_B$, represented by an inhibitor arc. Intuitively, this construction controls the evolution of the artifact by making sure that the cardinality of the relation with other artifacts is fulfilled.
(b) If tr contains an equality or inequality comparison to 1 on the number of instances of B:
  (i) In the case of an inequality comparison, a new place pl_B is added to $\overline{\mathcal{PN}}_{\mathtt{A}}$ and the arc (tr, pl') in $\overline{\mathcal{PN}}_{\mathtt{A}}$ is replaced by arc (tr, pl_B). A new invisible transition $continue_B$ is added to $\overline{\mathcal{PN}}_{\mathtt{A}}$ together with arcs: $(pl_B, continue_B)$, $(taken_B, continue_B)$, $(continue_B, taken_B)$ and $(continue_B, pl')$.
  (ii) Using the same place pl_B, we proceed similarly in the second case, with a new invisible transition $back_B$, with arcs $(pl_B, back_B)$ and $(back_B, pl')$ and a zero comparison from $taken_B$ to $back_B$, represented by an inhibitor arc.

[4] We introduced this possibility due to appearing often in specifications, but as the Figure 10 suggests, it is a modification of the one corresponding to the comparison to 0.

Figure 11 shows $\overline{\mathcal{PN}}_{user}$. The inclusion in $\overline{\mathcal{PN}}_{user}$ of the Petri nets that correspond to the activity diagrams to obtain the final Petri net $\mathcal{PN}_{user}$ is performed as before, and is not shown due to its complexity.

In summary, the presented transformation rules allow to generate a Petri net (possibly with inhibitor arcs) for each artifact, which apart from describing the artifact lifecycle, it encompasses the interaction cardinality constraints with other artifacts. The rules presented are clearly an initial proposal, which in spite of its simplicity, allowed us to deal with artifact-centric specifications like the one used in this paper.

We have presented the translations in an intuitive way, focusing on the describing precisely the important ideas of the translation, but not addressing their operationalization, an aspect that we leave as future work. In spite of this, the available tool support can be used to get more insight on the details not addressed in this paper.

5 Applying Conformance Checking to BAUML

Up to this point, we have shown how to translate the state machine diagram $\mathcal{S}$ and the activity diagrams $\mathcal{P}$ of a BAUML specification $\mathcal{B}= \langle \mathcal{M}, \mathcal{S}, \mathcal{P}, \mathcal{T} \rangle$ into a Petri net $\mathcal{PN}_{\mathtt{A}}$. Now, the obtained Petri net must be compared and contrasted with the real execution of the system $\mathcal{L}_{\mathtt{A}}$, where $\mathcal{L}_{\mathtt{A}}$ is the event log recorded by the

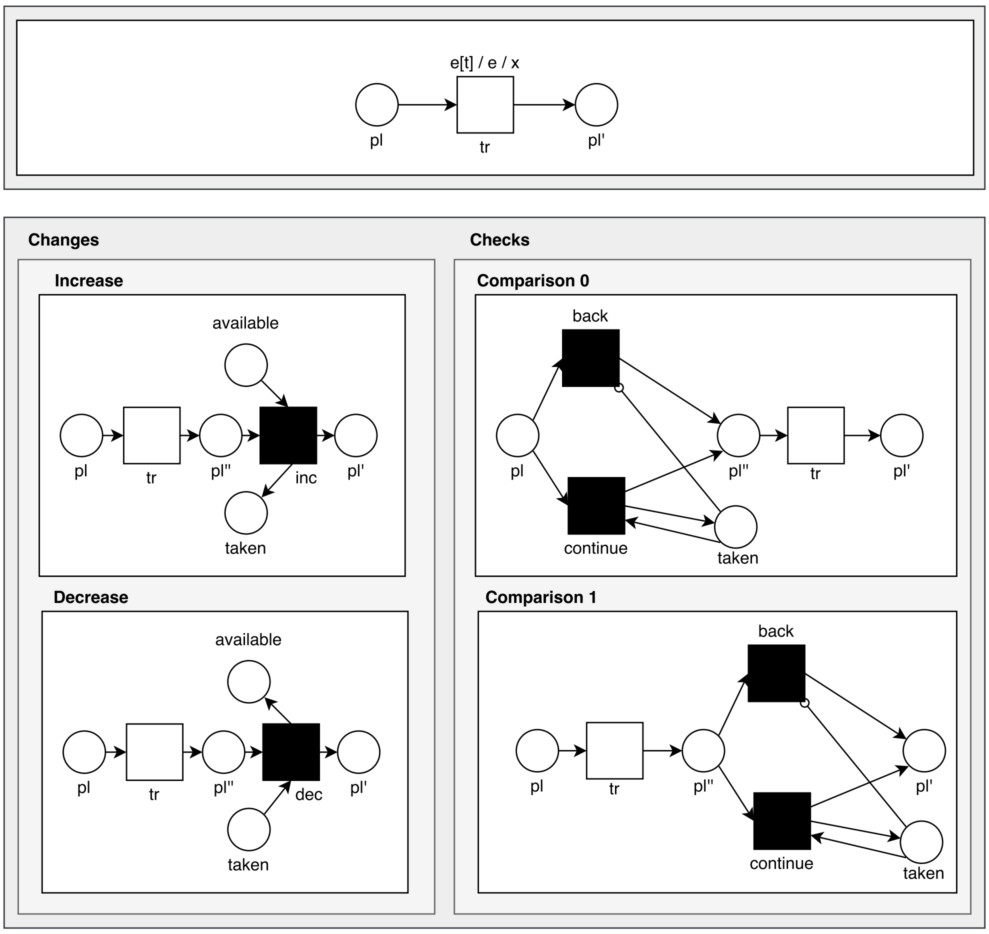


Fig. 10: Translation rules for artifacts with context-dependent behavior

system projected into the activities involved in the execution of artifact A, by means of *conformance checking* techniques. Then, these results can be mapped to the original BAUML model.

5.1 Introduction to Conformance Checking

Conformance checking techniques are able to identify main paths of the model executed by the system, and assess the quality of the model to describe the reality. But they also locate where exactly the mismatches between specification and reality are, in order to correct them on the specification or the implementation [36, 29]. This section shortly describes the alignment-based conformance checking. Next section will instantiate it to the problem of this paper.

Most of the state-of-the-art conformance checking techniques rely on the *alignment* of the event log and the Petri net model as a pre-processing step [5,29,10, 25]. In other words, the activities in a log trace $\sigma_{\mathtt{A}}$ must be linked to transitions of the Petri net $\mathcal{PN}_{\mathtt{A}}$ in the best-possible way, resulting on a so-called *alignment*. Figure 12 shows a possible alignment between the trace *CreateNewBicycle, AssigntoAnchorPoint, RequestBicycle, GetBicycle, ConfirmPick-Up, PlaceBicycleInAnchorPoint, ConfirmBicycleReturn* (i.e, the ordinary bicycle cycle), and the $\mathcal{PN}_{bike}$ shown in Figure 9. Notice that, for the sake of readability the names have been shortened. The upper row of the alignment represents the sequence of events, and the middle row the sequence of Petri net transitions linked to them. As it can be seen, in this case there is a direct way to align the log trace into the Petri net, i.e., each event of the log trace can be mimicked by the net (this is called a *synchronous move*).

When all the alignments of a log result in synchronous moves, it indicates that the specification model captures perfectly the behavior of the system. Moreover, the model can be enriched with the frequency of these moves, showing the most frequent paths executed in the system. However, there are other cases where, given a trace of the log, it is not possible to obtain an alignment with synchronous moves only. Let us

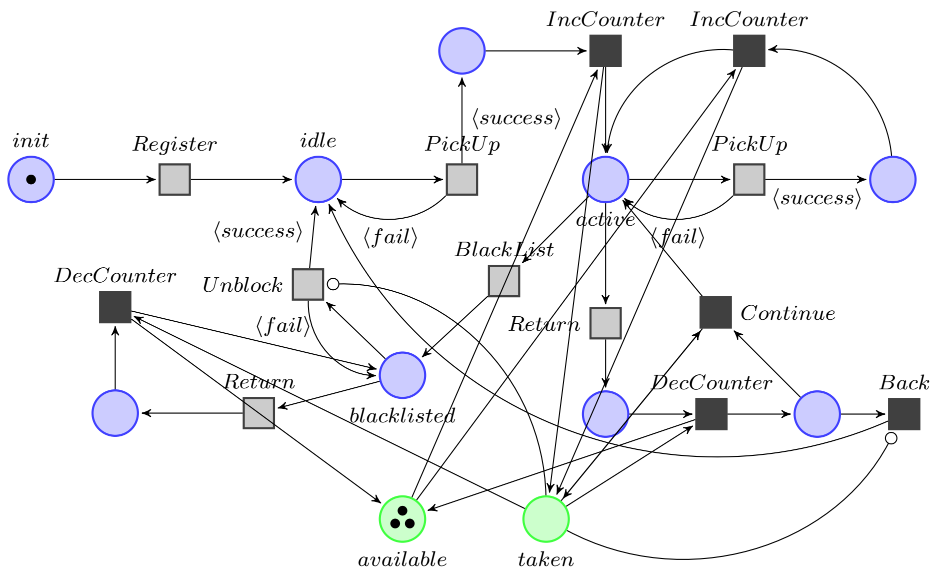


Fig. 11: The user artifact with counters (without refining activity diagram's transitions, and without using subscripts for related artifact instances on the silent transitions inserted).

c_r	a_s	r_{eq}	g_e	c_{op}	p_l	c_{br}
c_r	a_s	r_{eq}	g_e	c_{op}	p_l	c_{br}
0	0	0	0	0	0	0

Fig. 12: Alignment illustrating a perfect match between a typical bicycle picking recorded in the system and the model derived from the system specification.

consider the log trace *CreateNewBicycle, AssigntoAnchorPoint, RequestBicycle, GetBicycle, ConfirmPick-Up, PlaceBicycleInAnchorPoint, RequestBicycle*, i.e., a bicycle is picked up and returned, but the return is never manually confirmed by the user. A possible alignment is shown in Figure 13. In order to be mimicked by the model, the event *ConfirmBicycleReturn* is missing in the trace. This is indicated by the symbol $\gg$, calling it a *model move*. The detection of model moves in the alignments indicate deviations between the specification and the reality. In this example, the user forgets to manually check the return of the bicycle, an action advised to check that the system has correctly recorded the return, avoiding legal problems about lost bicycles. The last row of the alignment indicates the cost of the move according to a *cost function*, e.g., synchronous moves are preferred having a cost of 0, while the violation indicated by model move has a cost of 1. The costs of the alignments are used to assess numerically the quality of the specification with respect to the log. But more importantly, they indicate the exact location of the problems, and its possible causes.

c_r	a_s	r_{eq}	g_e	c_{op}	p_l	$\gg$	r_{eq}
c_r	a_s	r_{eq}	g_e	c_{op}	p_l	c_{br}	r_{eq}
0	0	0	0	0	0	1	0

Fig. 13: Alignment illustrating a case where the return of the bicycle is not confirmed by the used and specified.

Aligning techniques are also able to detect *log moves*, i.e., events in the trace that do not correspond with valid actions according to the model. Figure 14 shows an example of alignment with a log move. Given the trace *CreateNewBicycle, AssigntoAnchorPoint, RequestBicycle, GetBicycle, ConfirmPick-Up, PlaceBicycleInAnchorPoint, PlaceBicycleInAnchorPoint, ConfirmBicycleReturn*, a log move indicates that the action *PlaceBicycleInAnchorPoint* appears twice in the trace, when it should only appear once according to the specification. Log moves could indicate violations produced by errors in our system. For example, the system may be assigning two anchor points for the same returned bicycle, reducing the capacity of a station with *ghost* bicycles.

When an alignment has the lowest cost possible (i.e., there is no other possible alignment between the same trace and model with lowest cost), it is called *optimal*. Optimal alignments are preferred (being 0 the best sce-

c_r	a_s	r_{eq}	g_e	c_{op}	p_l	p_l	c_{br}
c_r	a_s	r_{eq}	g_e	c_{op}	p_l	$\gg$	c_{br}
0	0	0	0	0	0	1	0

Fig. 14: Alignment illustrating a malfunction of the system, where a bicycle is recorded twice being place on an anchor point.

nario possible), as they represent the best-way mapping between specification and log. In the literature, there exist algorithms to compute optimal alignments between Petri nets and logs in a wide range of scenarios (e.g., large models, different cost functions, ...), and approaches to assess the conformance from those alignments [5,29,2]. For more details on the alignment algorithms we refer the reader to those works.

Finally, Figure 15 shows the alignment for the trace *CreateNewBicycle, AssigntoAnchorPoint, RequestBicycle, GetBicycle, ReturntoAnchorPoint, ConfirmReturn, RequestBicycle, GetBicycle, ConfirmPick-Up*. As the alignment shows, the bicycle is picked up and returned immediately, indicating a problem in the bicycle. According to the specification, the bicycle should have been repaired (*RepairBicycle*), and eventually, anchored again to a station (*AnchorBicycleInAnchorPoint*), and marked as available (*MarkBicycleAsAvailable*). However, as the model moves show, the bicycle was never removed and repaired, making it possible for another user to take the broken bicycle again.

c_r	a_s	r_{eq}	g_e	r_{et}	c_{or}	$\gg$	$\gg$	$\gg$	r_{eq}	g_e	c_{op}
c_r	a_s	r_{eq}	g_e	r_{et}	c_{or}	r_{ep}	a_n	m_a	r_{eq}	g_e	c_{op}
0	0	0	0	0	0	1	1	1	0	0	0

Fig. 15: Alignment illustrating a broken bicycle returned to the ordinary circuit without the necessary reparation detailed in the specification.

5.2 Bringing up the results to the BAUML models

The results obtained previously focus into the alignment between a Petri net and a trace. However, in this work the Petri net acts as an intermediate model and, in order to close the circle, we are interested in projecting these results to the original BAUML models. We distinguish two kind of results to be transferred: those related to the artifact life-cycles, and those produced by mismatches in the cardinality relations among artifacts.

Projecting back to BAUML Artifact Life-cycle violations

There are two elements that need to be taken into consideration:

1. The activity diagram to which the task belongs, and
2. The context in which the task executes.

It is very easy to determine (1). Each labelled, visible transition in the Petri net corresponds exactly to a task in the UML activity diagrams. However, this information is not enough to locate the real problem in the initial models, as a certain activity diagram may execute under different circumstances or contexts, e.g. *Return Bicycle* in the case of *User*, which can take place when the user is *Blacklisted* or *Active*.

Therefore, in order to determine (2), it is only necessary to find the closest labelled places that correspond to states in the state machine diagram. This is quite straightforward to do as each place in the Petri net will either map univocally to exactly one state in the state machine diagram or to none. After performing this analysis, we will have obtained the context in which the activity diagram does not execute properly.

For instance, for the alignment in Figure 15, we know that tasks *RepairBicycle*, *AnchorBicycleInAnchorPoint* and *MarkBicycleAsAvailable* should have been executed. By analyzing the activity diagrams, we identify that all these tasks belong to the same activity diagram: *RecoverBicycle*.

Finding the context in which *RecoverBicycle* does not execute is easy in this case, because *RecoverBicycle* only appears once in the state machine diagram: the deviation detected is in the transition between states *Unusable* and *Available*.

This way, by analyzing (1) and (2) above, results can be projected back to the BAUML specification.

Projecting back to BAUML Artifact Context-dependency violations

Unlike control-flow violations, violations on the cardinality conditions among artifacts are not so straightforward to analyze and transfer back to the BAUML specification. For such cases, the model and log moves of the alignments need to be analyzed in order to detect patterns that pinpoint context-dependency violations. In this paper we propose two examples of patterns: *correction patterns* and *prevention patterns*. These patterns are based on the fact that alignments techniques provide an optimal (i.e., minimal cost) model-based explanation of the reality, which in some situations may suggests alternative ways of explaining what has happened in reality. Accordingly, the two patterns considered are not mutually exclusive: a deviation on the artifact cardinalities can sometimes be both explained either as a correction or prevention pattern.

Correction patterns are meant to provide an alternative to the observed trace so that the deviations on the cardinality relations between artifacts are avoided in the model explanation. These are based on the search for particular model moves. In practice, a correction pattern is a sequence of model moves artificially inserted in order to amend the cardinality relations between artifacts, e.g., before there is an increase that violates such conditions. For example, let us consider the trace *CreateNewBicycle, AssigntoAnchorPoint, RequestBicycle, GetBicycle, ConfirmPick-Up, RequestBicycle, GetBicycle, ConfirmPick-Up, RequestBicycle, GetBicycle, ConfirmPick-Up, RequestBicycle, GetBicycle, ConfirmPick-Up*, i.e., the malfuntioning system has allowed a user to succesfully pick-up 4 bikes, when the specification only allows for 3. Figure 16 shows the alignment resulting of the conformance check. Notice that the moves corresponding to invisible transitions (e.g., increase/decrease the number of bikes available) are denoted with a different color than the log/model moves, and have no cost penalty. As one could notice, between the third and forth increment of bikes in use, the alignment denotes a sequence of model moves *PlaceBicycleInAnchorPoint, ConfirmBicycleReturn*, i.e., an artificial return of a bike is performed in order to decrease the number of bikes in use before the forth bike is succesfully taken. Therefore, a segment of model moves that affect the corresponding available place indicate a correction pattern.

Prevention patterns are a symmetric perspective to correction patterns, focusing instead on log moves: an increment that violates a cardinality condition is never performed, and a sequence of log moves appears, in order to discard the events that must occur after that increment. For example, let us consider the same trace as before *CreateNewBicycle, AssigntoAnchorPoint, RequestBicycle, GetBicycle, ConfirmPick-Up, RequestBicycle, GetBicycle, ConfirmPick-Up, RequestBicycle, GetBicycle, ConfirmPick-Up, RequestBicycle, GetBicycle, ConfirmPick-Up, PlaceBicycleInAnchorPoint*, i.e., the user has picked-up 4 bikes before trying to return one of them, when the specification only allows for 3. The resulting alignment is shown in Figure 17. In this case, the fourth invisible move to increment the number of bikes in use never occurs, since it was not an available transition given the lack of tokens in the corresponding *available* place. And consequently, the next *PlaceBicycleInAnchorPoint* event in the log is considered a log move. Given the optimal nature of the alignment algorithm, a correction or prevention pattern is always determined by the lower penalty cost of the alignment. An analysis of the model could determine the less costly alternatives of model moves to appear in the correction patterns.

Finally, unlike artifact control-flow violations where conformance results could be used to correct the activity or state diagrams, context-dependent violations can be directly transferred to the class diagram. For instance, Figure 18 shows the *more than 3 bikes* violation detected in the previous alignments, highlighted in the annotation and cardinalities of the class diagram.

6 Implementation and Evaluation

In the previous sections we proposed a three-step approach to check conformance of BAUML specifications. These sections showed how a BAUML specification can be transformed into a Petri net, to then be checked against an event log execution using state-of-the-art conformance checking techniques. The ideas presented have been implemented (Section 6.1) and their potential has been evaluated using a plausible scenario (Section 6.2).

6.1 Implementation

The approach presented in this paper has been implemented and tested in the context of *ProM* framework [5]. ProM is an open-source Java framework for process mining and it is the most widely used tool for academia. Its plug-in architecture makes it possible to develop independent plug-ins for specific process mining tasks allowing them to interact with the rest of the existing plug-ins. In the context of the approach presented in this paper, two new plug-ins have been incorporated to the open repository of plug-ins of ProM, within the so-called *Specifact* package.[6] The package contains:

Import plug-in which import BAUML artifact specifications into ProM.
Conversion plug-in which convert an artifact specification in BAUML into a Petri net with potentially inhibitor arcs (cf, Figure 19).

As is it previously mentioned, these plug-ins are combined with other existing plug-ins to analyze the conformance of the specification. In particular, the plug-ins useful for this purpose are:

[5] `http://www.promtools.org/`
[6] `https://svn.win.tue.nl/trac/prom/browser/Packages/Specifact`

c_u	r_{eq}	g_e	c_{op}	$\gg$	r_{eq}	g_e	c_{op}	$\gg$	r_{eq}	g_e	c_{op}	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	r_{eq}	g_e	c_{op}	$\gg$
c_u	r_{eq}	g_e	c_{op}	i_{nc}	r_{eq}	g_e	c_{op}	i_{nc}	r_{eq}	g_e	c_{op}	i_{nc}	p_l	c_{br}	d_{ec}	c_{on}	r_{eq}	g_e	c_{op}	i_{nc}
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0

Fig. 16: Alignment illustrating a correction pattern case where a bike is artificially returned using model moves to denote the violation of the number of bikes in use condition.

c_u	r_{eq}	g_e	c_{op}	$\gg$	r_{eq}	g_e	c_{op}	$\gg$	r_{eq}	g_e	c_{op}	$\gg$	r_{eq}	g_e	c_{op}	p_l
c_u	r_{eq}	g_e	c_{op}	i_{nc}	r_{eq}	g_e	c_{op}	i_{nc}	r_{eq}	g_e	c_{op}	i_{nc}	r_{eq}	g_e	c_{op}	$\gg$
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Fig. 17: Alignment illustrating a prevention pattern case where the increment of number of bikes in use is prevented by means of a log move.

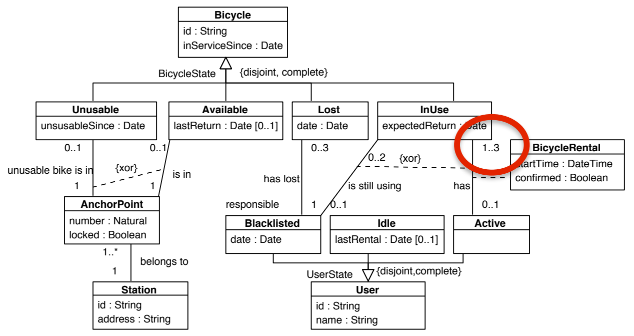


Fig. 18: Context-dependent cardinality violations highlighted in the Bicing class diagram of Figure 2.

Import plug-ins which import event logs in different formarts (e.g., XES or MXML) into ProM.
Export plug-ins which allow to export and store the resulting Petri net in some standard Petri net format (e.g., PNML).
Conformance plug-ins which implement the different conformance techniques, such as the alignments algorithms over Petri nets (e.g., *Replay a Log on Petri Net for Conformance Analysis* plug-in used in the evaluation).
Visualization plug-ins which graphically visualize the Petri net, the alignments, and the conformance results.
Conversion plug-ins which convert the resulting Petri net to other modelling notations such as Process trees or BPMN for further analysis out of the scope of this paper.
Utils plug-ins that includes a wide range of useful tools, for example to filter events of a log.

Finally, in order to facilitate the definition of the BAUML models for the user, we use Visual Paradigm[7] to create the various diagrams in the framework. Visual Paradigm is a commercial CASE tool which offers a free community edition for non-commercial purposes. From these diagrams it is possible to export an XML file which can then be processed and translated into a Petri net by our plug-ins.

6.2 Evaluation

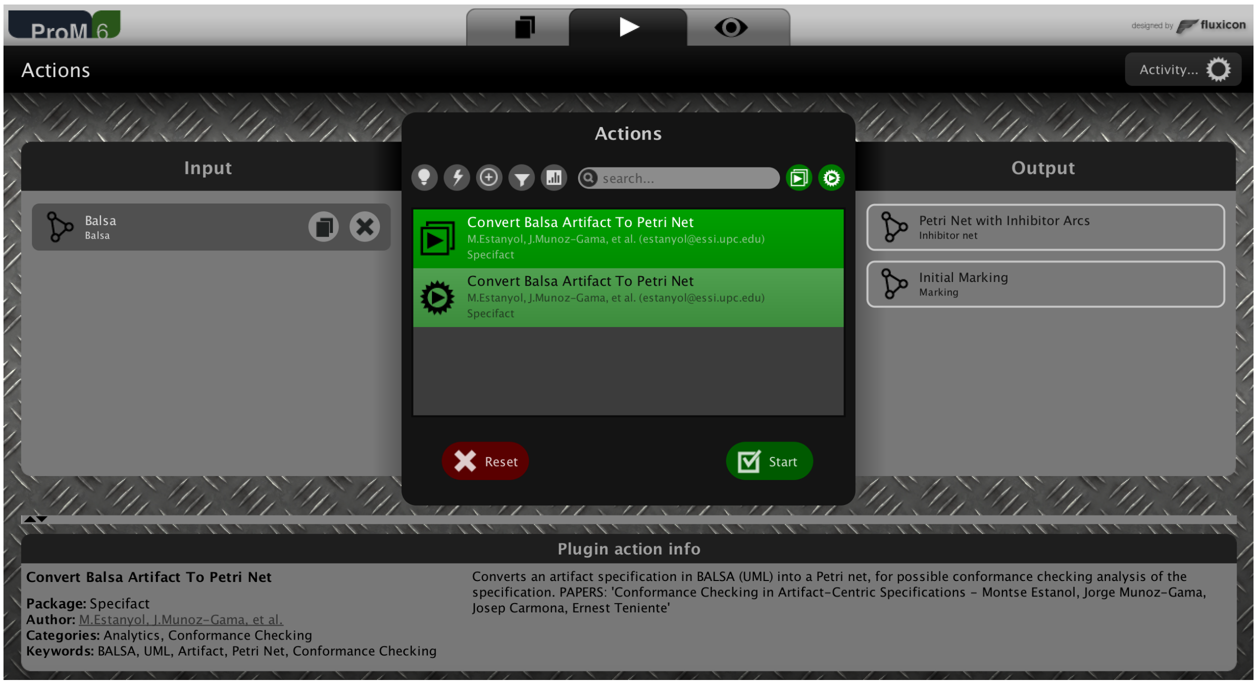
In the Introduction we presented the need for analyzing whether a real implementation satisfies its corresponding artifact-centric specification. In this section we illustrate how our approach can indeed fulfill this need, using conformance checking techniques. In order to do that, we use the *Bicing* case used as a running example in this paper.

Figure 20 shows a screenshot of the activity diagrams in our BAUML specification in Visual Paradigm. The whole model can be exported into an XML file (see Figure 21).

Figure 22 shows the results of importing the BAUML specification (in an XML file) into ProM using the importer designed for such task, and applying the plug-in *Convert Balsa Artifact to Petri net* to the *bicycle* artifact. The result is a Petri net without inhibitor arcs due to the artifact context-independent behavior, the same net as the one shown in Figure 9.

On the other hand, Figure 23 partially shows a fragment of the system log. The log used in this evaluation contains 8031 cases, and 84514 events, over 18 different activities. The format used for the log is *XES*, a

[7] `https://www.visual-paradigm.com`



Convert Balsa Artifact To Petri Net
M.Estanyol, J.Munoz-Gama, et al. (estanyol@essi.upc.edu)
Specifact

Convert Balsa Artifact To Petri Net
M.Estanyol, J.Munoz-Gama, et al. (estanyol@essi.upc.edu)
Specifact

Convert Balsa Artifact To Petri Net

Package: Specifact
Author: M.Estanyol, J.Munoz-Gama, et al.
Categories: Analytics, Conformance Checking
Keywords: BALSA, UML, Artifact, Petri Net, Conformance Checking

Converts an artifact specification in BALSA (UML) into a Petri net, for possible conformance checking analysis of the specification. PAPERS: 'Conformance Checking in Artifact-Centric Specifications - Montse Estanol, Jorge Munoz-Gama, Josep Carmona, Ernest Teniente'

Fig. 19: A screenshot of ProM showing the plug-ins to convert a BAUML object into a Petri net with inhibitor arcs with its initial marking. The top plug-in allows for a graphical configuration of the parameters such as the selection of the artifact to convert, while the bottom one takes all the paremeters by default.

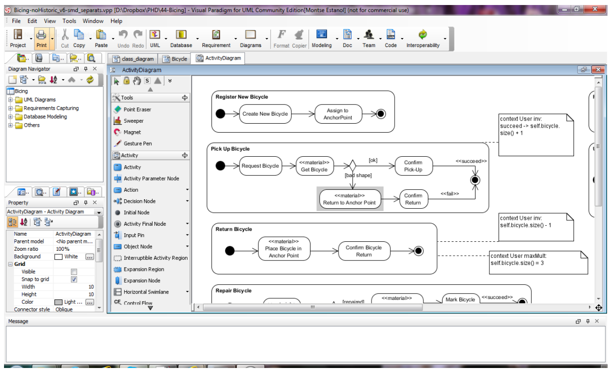


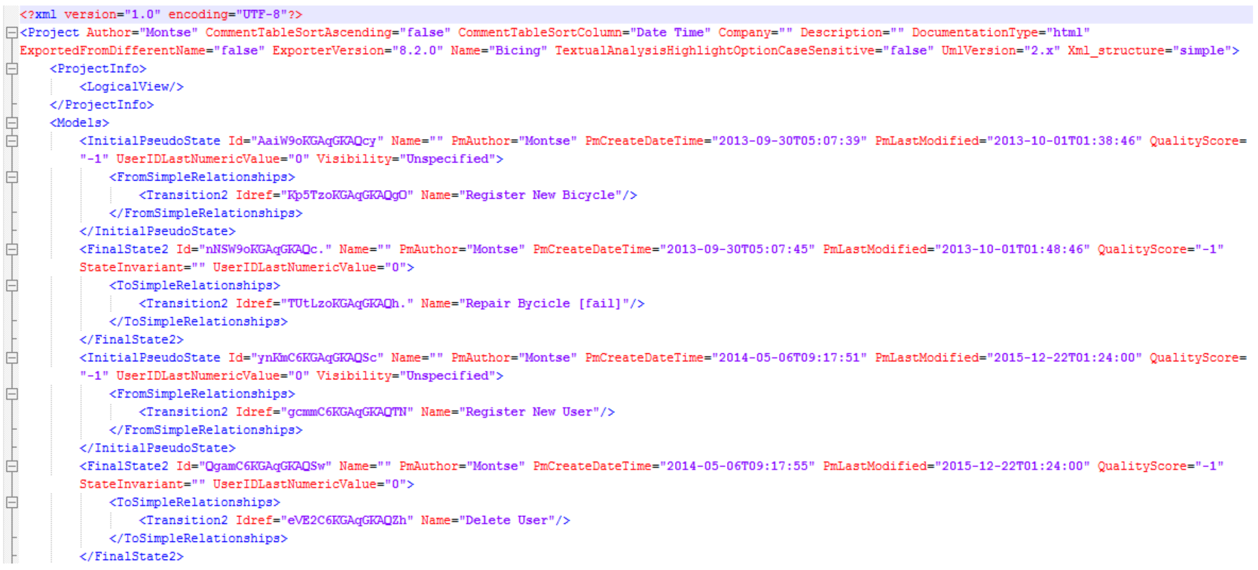
Fig. 20: Screenshot of our BAUML model in Visual Paradigm

XML-based format for the interchange of event log data between tools and application domains, approved by the IEEE as the Standard for eXtensible Event Stream (XES) for Achieving Interoperability in Event Logs and Event Streams (1849-2016).[8] The log only includes those events in the system related with the activities in the artifact bicycle.

The Petri net and the log are used to check the conformance between specification and system. Figures 24 and 25 show the conformance results obtained using the *Replay a Log on Petri Net for Conformance Analysis* plug-in. The plug-in performs the alignment between the log traces and the Petri net, and then the results are displayed in different interactive views for a deeper analysis. In Figure 24 the alignment results are used to enhance the original model. This view allow us to point out the *highroads* of our process, i.e., the parts of our process with more executions are colored in dark blue, while the more infrequent parts are colored in a light blue. In this example, we conclude that most of the executions involve the ordinary pick-up/return cycle of the bicycle, while the repair/destroy/lost parts of the model are infrequent.

This view also allows us to analyze the alignment mismatches (move and log moves) in an aggregated way. On the one hand, places involved in these violations are colored in yellow. On the other hand, transitions with model moves display a colored bar indicating the ratio of model moves aligned with respect to the total of synchronous moves. Figure 26 shows a zoomed-in view of the activity *ConfirmBicycleReturn*, next to the legend of the colored bar. From the figure we conclude that, although this is not the general case, the number of instances where a user does not confirm the return of the bicycle is not negligible, i.e., 2480 out of 10070 are not confirmed. This systemic problem may lead to an increase of wrongly reported lost bicycles, a factor that can be mitigated including modifications on the system

[8] http://www.xes-standard.org/



```xml
<?xml version="1.0" encoding="UTF-8"?>
<Project Author="Montse" CommentTableSortAscending="false" CommentTableSortColumn="Date Time" Company="" Description="" DocumentationType="html"
ExportedFromDifferentName="false" ExporterVersion="8.2.0" Name="Bicing" TextualAnalysisHighlightOptionCaseSensitive="false" UmlVersion="2.x" Xml_structure="simple">
    <ProjectInfo>
        <LogicalView/>
    </ProjectInfo>
    <Models>
        <InitialPseudoState Id="AaiW9oKGAqGKAQcy" Name="" PmAuthor="Montse" PmCreateDateTime="2013-09-30T05:07:39" PmLastModified="2013-10-01T01:38:46" QualityScore=
        "-1" UserIDLastNumericValue="0" Visibility="Unspecified">
            <FromSimpleRelationships>
                <Transition2 Idref="Kp5TzoKGAqGKAQgO" Name="Register New Bicycle"/>
            </FromSimpleRelationships>
        </InitialPseudoState>
        <FinalState2 Id="nNSW9oKGAqGKAQc." Name="" PmAuthor="Montse" PmCreateDateTime="2013-09-30T05:07:45" PmLastModified="2013-10-01T01:48:46" QualityScore="-1"
        StateInvariant="" UserIDLastNumericValue="0">
            <ToSimpleRelationships>
                <Transition2 Idref="TUtLzoKGAqGKAQh." Name="Repair Bycicle [fail]"/>
            </ToSimpleRelationships>
        </FinalState2>
        <InitialPseudoState Id="ynKmC6KGAqGKAQSc" Name="" PmAuthor="Montse" PmCreateDateTime="2014-05-06T09:17:51" PmLastModified="2015-12-22T01:24:00" QualityScore=
        "-1" UserIDLastNumericValue="0" Visibility="Unspecified">
            <FromSimpleRelationships>
                <Transition2 Idref="gcmmC6KGAqGKAQTN" Name="Register New User"/>
            </FromSimpleRelationships>
        </InitialPseudoState>
        <FinalState2 Id="QgamC6KGAqGKAQSw" Name="" PmAuthor="Montse" PmCreateDateTime="2014-05-06T09:17:55" PmLastModified="2015-12-22T01:24:00" QualityScore="-1"
        StateInvariant="" UserIDLastNumericValue="0">
            <ToSimpleRelationships>
                <Transition2 Idref="eVE2C6KGAqGKAQZh" Name="Delete User"/>
            </ToSimpleRelationships>
        </FinalState2>
```

Fig. 21: Fragment of the XML which results from exporting our BAUML model

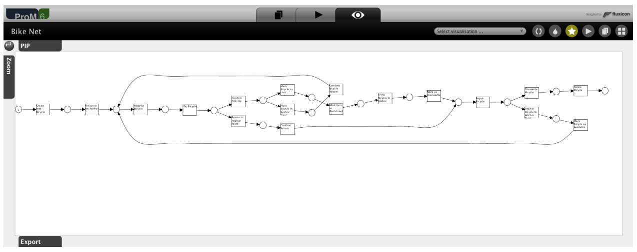
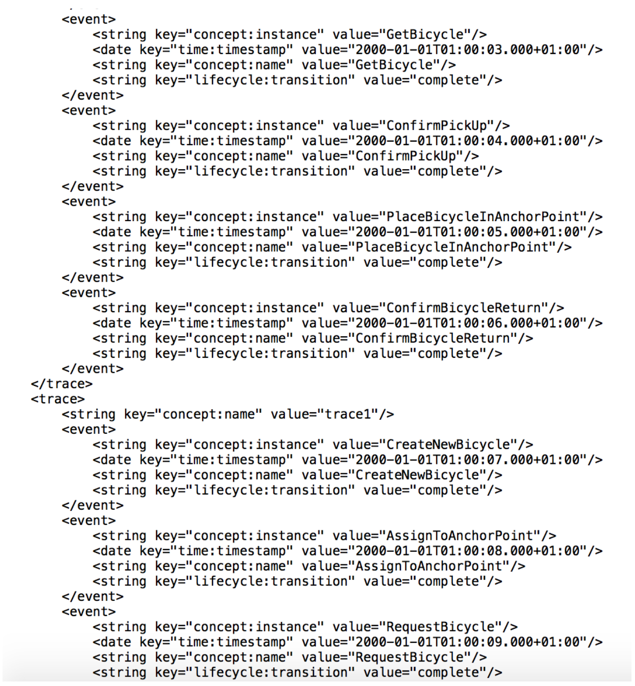


Fig. 22: Net result of converting the artifact-specification bicycle into a Petri net.



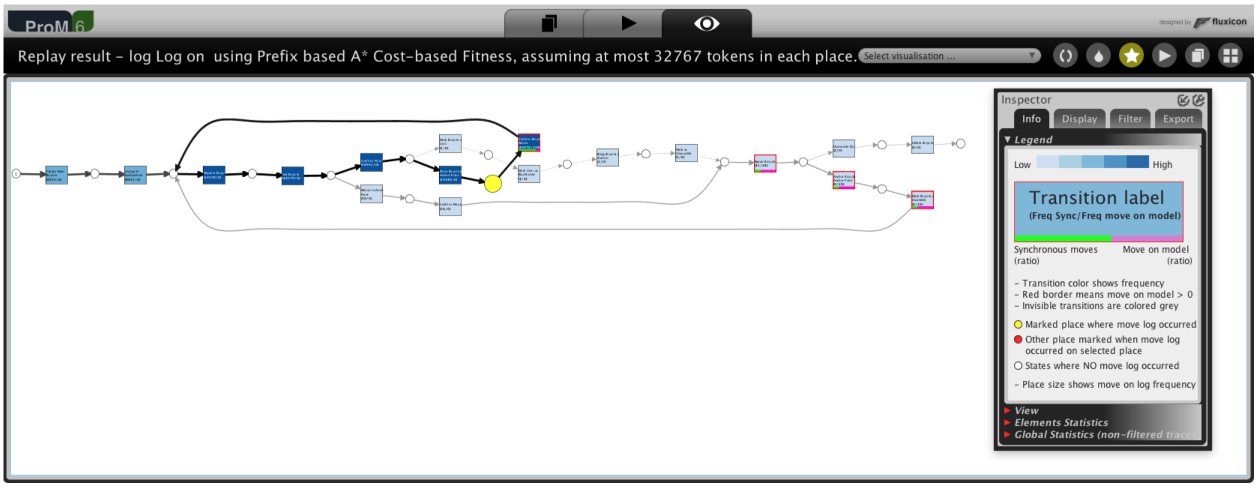
```xml
    <event>
        <string key="concept:instance" value="GetBicycle"/>
        <date key="time:timestamp" value="2000-01-01T01:00:03.000+01:00"/>
        <string key="concept:name" value="GetBicycle"/>
        <string key="lifecycle:transition" value="complete"/>
    </event>
    <event>
        <string key="concept:instance" value="ConfirmPickUp"/>
        <date key="time:timestamp" value="2000-01-01T01:00:04.000+01:00"/>
        <string key="concept:name" value="ConfirmPickUp"/>
        <string key="lifecycle:transition" value="complete"/>
    </event>
    <event>
        <string key="concept:instance" value="PlaceBicycleInAnchorPoint"/>
        <date key="time:timestamp" value="2000-01-01T01:00:05.000+01:00"/>
        <string key="concept:name" value="PlaceBicycleInAnchorPoint"/>
        <string key="lifecycle:transition" value="complete"/>
    </event>
    <event>
        <string key="concept:instance" value="ConfirmBicycleReturn"/>
        <date key="time:timestamp" value="2000-01-01T01:00:06.000+01:00"/>
        <string key="concept:name" value="ConfirmBicycleReturn"/>
        <string key="lifecycle:transition" value="complete"/>
    </event>
</trace>
<trace>
    <string key="concept:name" value="trace1"/>
    <event>
        <string key="concept:instance" value="CreateNewBicycle"/>
        <date key="time:timestamp" value="2000-01-01T01:00:07.000+01:00"/>
        <string key="concept:name" value="CreateNewBicycle"/>
        <string key="lifecycle:transition" value="complete"/>
    </event>
    <event>
        <string key="concept:instance" value="AssignToAnchorPoint"/>
        <date key="time:timestamp" value="2000-01-01T01:00:08.000+01:00"/>
        <string key="concept:name" value="AssignToAnchorPoint"/>
        <string key="lifecycle:transition" value="complete"/>
    </event>
    <event>
        <string key="concept:instance" value="RequestBicycle"/>
        <date key="time:timestamp" value="2000-01-01T01:00:09.000+01:00"/>
        <string key="concept:name" value="RequestBicycle"/>
        <string key="lifecycle:transition" value="complete"/>
```

Fig. 23: Partial view of the system log, only including the activities of the artifact bicycle.

(e.g., a red led indicating a warning until the return is manually confirmed by the user using the bicing card).

The view shown in Figure 25 presents the results of the alignment in a more fine-grained level of detail. Each sequence of triangles represent the alignment for one case variant. The triangles are colored depending on the type of move: synchronous moves (green), log moves (yellow), and model moves (purple). The view is interactive, and placing the mouse over each triangle gives you the activity name it represents. The plug-in also provides quantitative information about the conformance, such as metrics and statistics per trace and per log.

Figure 27 shows a close-up of one alignment. The alignment indicates a bicycle that is picked up and immediately returned (indicating a problem), but is immediately picked-up and returned several times until is is finally repaired. The example is similar to the one shown in Figure 15, where the repair activities are never performed (i.e., model moves) before the bicycle is picked-up again. These discrepancies between specification and system indicate a systemic problem that needs to be addressed, e.g., the protocols for repairing the bicycles are not being properly followed. It may also indicate that the specification is not suitable for the reality and it requires an update, e.g., the bicycle is available for a certain number of tries until it is reported as broken, or the specification needs to include an additional activity between the return and the repair indicating that the bike should be set to *unavailable*.



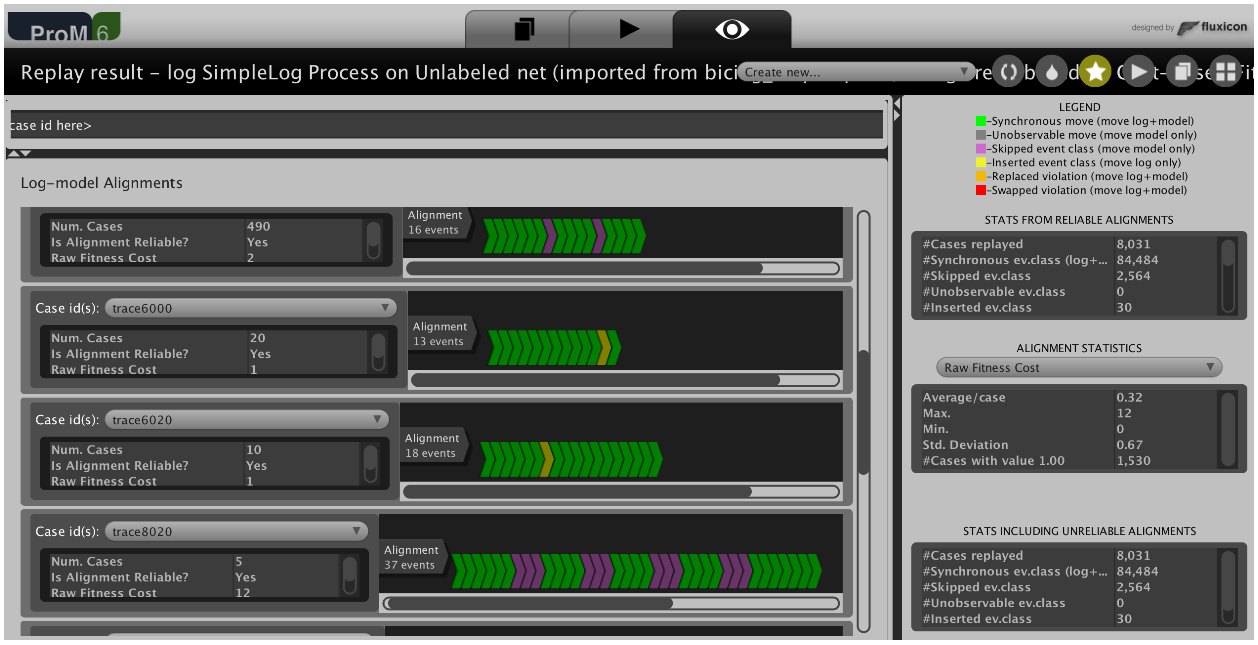
Replay result – log Log on using Prefix based A* Cost-based Fitness, assuming at most 32767 tokens in each place.

- Transition color shows frequency
- Red border means move on model > 0
- Invisible transitions are colored grey

○ Marked place where move log occurred
● Other place marked when move log occurred on selected place
○ States where NO move log occurred

- Place size shows move on log frequency

Fig. 24: Conformance checking results over the bicycle artifact Petri net, showing the *highroads* of the system execution.



Replay result – log SimpleLog Process on Unlabeled net (imported from bici

Num. Cases	490
Is Alignment Reliable?	Yes
Raw Fitness Cost	2

Case id(s): trace6000

Num. Cases	20
Is Alignment Reliable?	Yes
Raw Fitness Cost	1

Case id(s): trace6020

Num. Cases	10
Is Alignment Reliable?	Yes
Raw Fitness Cost	1

Case id(s): trace8020

Num. Cases	5
Is Alignment Reliable?	Yes
Raw Fitness Cost	12

LEGEND
🟩–Synchronous move (move log+model)
▨–Unobservable move (move model only)
🟪–Skipped event class (move model only)
🟨–Inserted event class (move log only)
🟧–Replaced violation (move log+model)
🟥–Swapped violation (move log+model)

STATS FROM RELIABLE ALIGNMENTS

#Cases replayed	8,031
#Synchronous ev.class (log+...	84,484
#Skipped ev.class	2,564
#Unobservable ev.class	0
#Inserted ev.class	30

ALIGNMENT STATISTICS

Raw Fitness Cost

Average/case	0.32
Max.	12
Min.	0
Std. Deviation	0.67
#Cases with value 1.00	1,530

STATS INCLUDING UNRELIABLE ALIGNMENTS

#Cases replayed	8,031
#Synchronous ev.class (log+...	84,484
#Skipped ev.class	2,564
#Unobservable ev.class	0
#Inserted ev.class	30

Fig. 25: Alignment view reflecting the specific moves for each trace of the log.

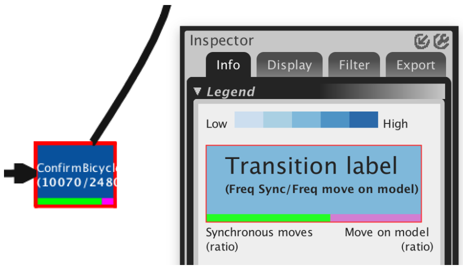


Fig. 26: Figure showing the non-negligible proportion of instances where the returning of the bike is not confirmed.

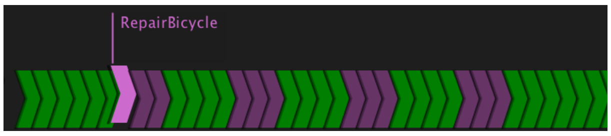


Fig. 27: Example of alignment showing bicycle picked up and immediately returned until it is finally repaired.

Figure 28 shows a similar case, where the system records twice that a certain bicycle is being returned, denoted by a log move of *PlaceBicycleInAnchorPoint* just after a synchronous move of the same activity. This problem denotes an issue with the system implementation that could lead to an sub-optimal use of the stations, producing ghosts or duplicate bicycles, difficult to track.

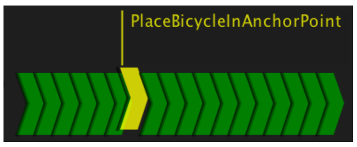


Fig. 28: Example of alignment showing a miss-recording on the system, assigning twice an anchor point for the same returned bicycle.

Although these results are obtained over the Petri net, it is easy to transfer them over to the original model, as each of the places in the Petri net corresponds to a task. For instance, the issue with a bicycle being returned twice, shown in Figure 28, reflects in the activity diagram of *Return Bicycle*, shown in Figure 29.

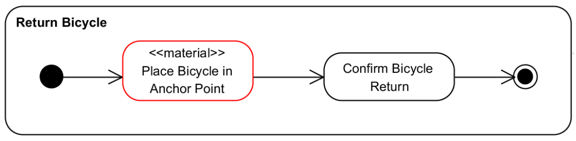


Fig. 29: Example of alignment showing a miss-recording on the system, assigning twice an anchor point for the same returned bicycle.

Similarly, the conformance analysis can be performed over the artifacts with context-dependent behavior such as the *user* artifact. Figure 30 shows the resulting Petri net using the proposed approach. This net is then aligned with the log to detect possible deviations. Figure 31 shows the case where the system records a user requesting a fourth bicycle while the user already has three bicycles in use. The fourth consecutive request is represented by a log move, denoting a problem with the system implementation that should not allow for the possibility of requesting another bicycle. Notice that, although the detection of misalignment using conformance checking is an automatic task, the explanation and contextualization of the misalignment requires is a human interpretation. However, conformance checking techniques aid on that interpretation, detecting and highlighting the anomalies and the elements involved.

7 Related Work

The work we present here is part of two research areas: conformance checking and artifact-centric business process models. For this reason, the first part of this section gives an overview of the field of conformance checking and the second part analyzes process-centric and artifact-centric proposals for business process modeling. At the end, we deal with conformance checking applied to artifact-centric business process models, which is what we do in this work.

Conformance checking emerged as the area of Process Mining that compares existing process models with actual observations of the process execution represented as an event log in order to assess their quality [1]. The seminal work of [36] presented the conformance between a model and a log as a multi-dimensional quality, and proposed best-effort metrics to asses such comparison. Recently, more sophisticate conformance definitions have been proposed, aligning both executed and modeled behaviors, and providing an optimal assessment of the conformance [5]. Conformance based on aligning are considered the state-of-the-art methods, and have been adapted to specific scenarios, such as large processes [30,2], genetic algorithms [9], multi-perspectives [27], and partial order [26]. Most of conformance checking techniques focused on the control flow perspective of the processes [1]. In spite of this, techniques that take into account further dimensions like data or resources have appeared recently, which can represent a first step towards enhancing the expressive power of conformance checking [25].

In terms of representing the business processes, we distinguish between process-centric and artifact-centric approaches. Process-centric approaches focus on the control-flow of the process, which tend to use languages such as BPMN, YAWL, UML Activity Diagrams or Workflow nets [39] for representing the processes. In the case of BPMN and UML activity diagrams there exist techniques to translate these models into Petri nets [14, 38]. On the other hand YAWL and WorkFlow nets are already based on Petri nets [3,4]. However, there are no specific techniques to consider the data when applying conformance checking to the equivalent Petri nets of these models.

Artifact-centric approaches offer a variety of models to represented the processes and their required data. Many of the existing alternatives such as [6,12,21], are grounded on logic which makes them formal and unambiguous, but impractical from the point of view of business managers and developers. Other alternatives, such as GSM [13,37], using a combination of models based on UML and OCL [11,15], or artifact union graphs [8] provide a more user-friendly representation of the models.

However, none of these works actually deal with the problem of conformance between a process model and its execution logs. In particular [8,37,11,15,6,12,21,28] deal with different aspects of the verification of these models before they are put into practice, to ensure that

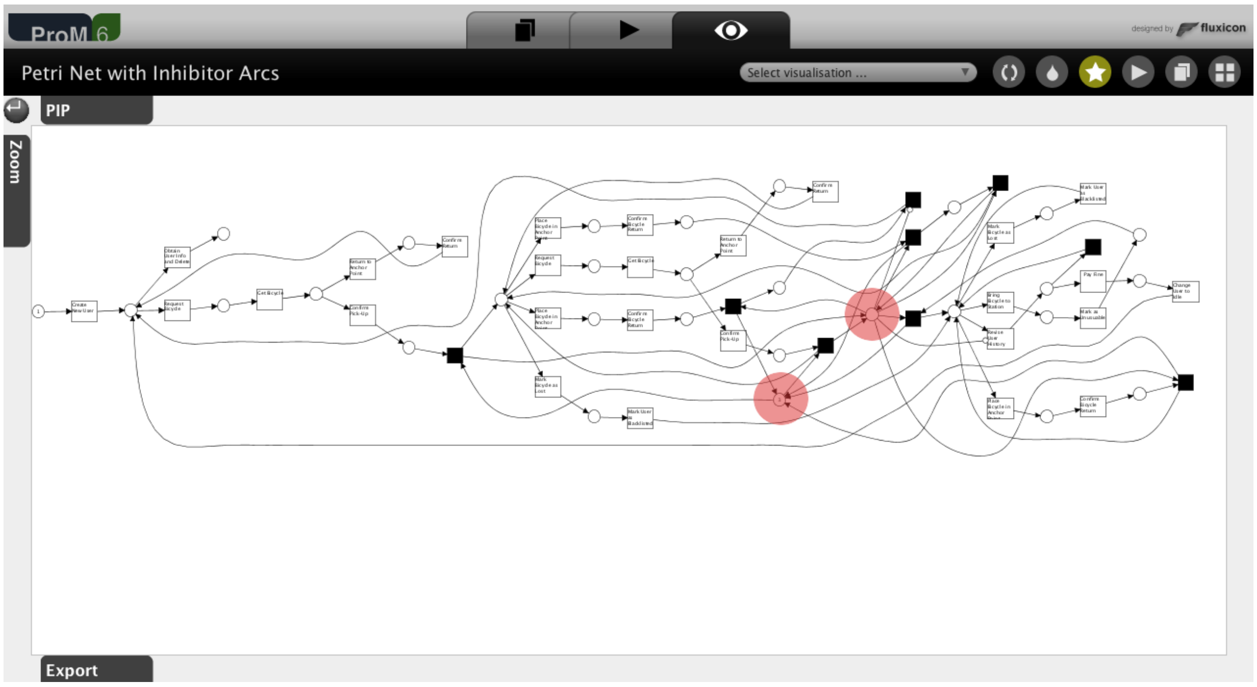


Fig. 30: Resulting Petri net for the *user* artifact, highlighting the two places *available* and *taken* included to control the maximum number of bicycles per user.

there are no errors and that they fulfill the business requirements.

To the best of our knowledge, the approaches in [19, 18] are the only ones in facing the problem of conformance checking for artifact-centric specifications. The artifact-centric process models considered are proclets, which are Petri nets enriched with a set of channels that coordinate the different relations artifacts have on their corresponding lifecycle. Interestingly, the problem of conformance for these tailored specifications is mapped to the classical conformance checking instance so that available tools for traditional conformance checking can be used. However, as already commented as one of the core motivations of this work, it is not yet clear how proclets can be obtained (initial attemps for this important step were recently presented in [34]), and their manual creation may represent a burden for non Petri net oriented users.

It is also worth mentioning that by using the BAUML model for specifying business processes we make a step forward to bridge the gap between models with a high level of abstraction, which are more intuitive and understandable for business people, and notations which are very formal (such as Petri nets) but usually impractical from the point of view of the business.

8 Conclusions and further work

In this paper we present a framework for conformance checking of BAUML artifact-centric specifications. To do so, we show how the models (i.e. the state machine and activity diagrams) in the framework can be translated into a Petri net so that existing conformance checking techniques can be applied. This is one of the contributions of the paper.

Secondly, in contrast to most existing works that perform conformance checking, we are able to incorporate into the Petri nets some of the data constraints that appear in the state machine diagram, and which limit the allowed executions of the tasks that are part of our models. We do so by considering the creation and deletion of associations between artifacts, specified in the tasks themselves, and the conditions restricting the number of artifacts which can be related to other artifacts, when carrying out certain transitions in the state machine diagram.

Finally, we have implemented a plug-in within the open-source ProM framework. This plug-in is able to extract the BAUML models from an XML file generated by Visual Paradigm and translate them into the appropriate Petri nets. Then, by using existing conformance checking plug-ins, we are able to detect the deviations between the initial BAUML models and the executions. Moreover, a detailed real-life-like case study is reported.

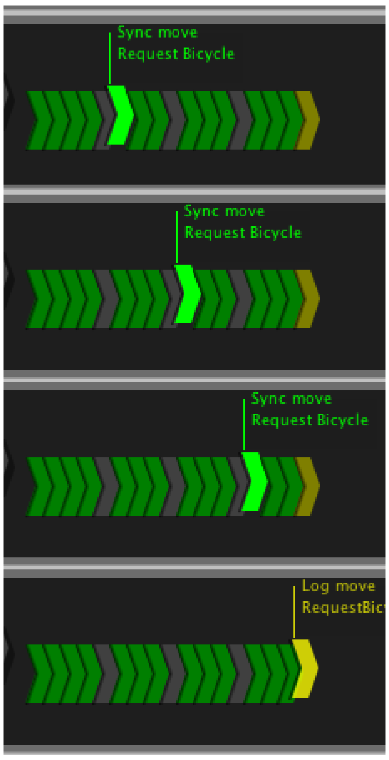


Fig. 31: Example of an alignment showing a flaw on the system, allowing a user to request a fourth bicycle while the user already has three in use.

To sum up, we believe this work represents an important step towards bridging the gap between current process mining algorithms and their adoption in settings like software engineering. By extending the analysis capabilities to other dimensions like the artifact lifecycles, the information provided by conformance checking techniques can identify important failures hidden in a specification. Although our work is based on the BAUML framework, using a set of UML and OCL models, any other notation which can be formalized as BAUML could also be used.

As further work, we plan to apply the proposed approach on a real case study with specific domain questions. Notice that this will necessarily require the design of a system based on BAUML, for its later implementation, deployment, and the collection of execution data for the analysis.

Acknowledgements This work is partially supported by *FONDECYT Iniciación 11170092* and *CONICYT Apoyo a la Formación de Redes Internacionales Para Investigadores en Etapa Inicial REDI170136*; by the Spanish Ministerio de Economía y Competitividad, under projects TIN2017-87610-R and TIN2017-86727-C2-1-R; and by the Secreteria d'Universitats i Recerca de la Generalitat de Catalunya, under 2017-SGR-1749.

References

1. van der Aalst, W.M.P.: Process Mining: Discovery, Conformance and Enhancement of Business Processes (2011)
2. van der Aalst, W.M.P.: Decomposing petri nets for process mining: A generic approach. Distributed and Parallel Databases **31**(4), 471–507 (2013)
3. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: yet another workflow language. Inf. Syst. **30**(4), 245–275 (2005)
4. van der Aalst, W.M.P., et al.: Soundness of workflow nets: classification, decidability, and analysis. Formal Aspects of Computing **23**(3), 333–363 (2011)
5. Adriansyah, A.: Aligning Observed and Modeled Behavior. Ph.D. thesis, Eindhoven University of Technology (2014)
6. Bagheri Hariri, B., et al.: Verification of relational data-centric dynamic systems with external services. In: PODS, pp. 163–174. ACM (2013)
7. Basu, S., et al. (eds.): Service-Oriented Computing - 11th International Conference, ICSOC 2013, *LNCS*, vol. 8274. Springer (2013)
8. Borrego, D., Gasca, R.M., López, M.T.G.: Automating correctness verification of artifact-centric business process models. Information & Software Technology **62**, 187–197 (2015)
9. Buijs, J.C.A.M.: Flexible Evolutionary Algorithms for Mining Structured Process Models. Ph.D. thesis, Eindhoven University of Technology (2014)
10. Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: Quality dimensions in process discovery: The importance of fitness, precision, generalization and simplicity. Int. J. Cooperative Inf. Syst. **23**(1) (2014)
11. Calvanese, D., Montali, M., Estañol, M., Teniente, E.: Verifiable UML artifact-centric business process models. In: CIKM 2014, pp. 1289–1298. ACM (2014)
12. Damaggio, E., Deutsch, A., Vianu, V.: Artifact systems with data dependencies and arithmetic. ACM Trans. Database Syst. **37**(3), 22 (2012)
13. Damaggio, E., Hull, R., Vaculín, R.: On the equivalence of incremental and fixpoint semantics for business artifacts with Guard Stage Milestone lifecycles. Inf. Syst. **38**(4), 561 – 584 (2013). Special section on BPM 2011 conference
14. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Information & Software Technology **50**(12), 1281–1294 (2008)
15. Estañol, M., Sancho, M., Teniente, E.: Verification and validation of UML artifact-centric business process models. In: J. Zdravkovic, M. Kirikova, P. Johannesson (eds.) CAiSE 2015, *LNCS*, vol. 9097, pp. 434–449. Springer (2015)
16. Estañol, M., Sancho, M.R., Teniente, E.: Reasoning on UML data-centric business process models. In: Basu et al. [7], pp. 437–445
17. Estañol, M., Sancho, M.R., Teniente, E.: Ensuring the semantic correctness of a BAUML artifact-centric BPM. Information and Software Technology **93**, 147–162 (2018)
18. Fahland, D., de Leoni, M., van Dongen, B.F., van der Aalst, W.M.P.: Conformance checking of interacting processes with overlapping instances. In: S. Rinderle-Ma, F. Toumani, K. Wolf (eds.) BPM 2011. Proceedings, *LNCS*, vol. 6896, pp. 345–361. Springer (2011)
19. Fahland, D., Leoni, M.D., van Dongen, B.F., van der Aalst, W.M.P.: Behavioral conformance of artifact-centric process models. In: W. Abramowicz (ed.) BIS 2011, *LNBIP*, vol. 87, pp. 37–49. Springer (2011)

20. Fahland, D., et al.: Checking behavioral conformance of artifacts. Tech. Rep. BPM-11-07, BPM Center (2011)
21. Gerede, C.E., Su, J.: Specification and verification of artifact behaviors in business process models. In: B.J. Krämer, K.J. Lin, P. Narasimhan (eds.) ICSOC, *LNCS*, vol. 4749, pp. 181–192. Springer (2007)
22. Hull, R.: Artifact-centric business process models: Brief survey of research results and challenges. In: R. Meersman, Z. Tari (eds.) OTM 2008, *LNCS*, vol. 5332, pp. 1152–1163. Springer (2008)
23. ISO: ISO/IEC 19505-2:2012 - OMG UML superstructure 2.4.1 (2012). Available at: `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=52854`
24. de Leoni, M., Maggi, F.M., van der Aalst, W.M.P.: An alignment-based framework to check the conformance of declarative process models and to preprocess event-log data. Inf. Syst. **47**, 258–277 (2015). DOI 10.1016/j.is.2013.12.005. URL `http://dx.doi.org/10.1016/j.is.2013.12.005`
25. Leoni, M.D., Aalst, W.M.P.V.D., Dongen, B.F.V.: Data- and Resource-Aware Conformance Checking of Business Processes. In: Business Information Systems, vol. 87, pp. 48–59. Springer (2012)
26. Lu, X., Fahland, D., van der Aalst, W.M.P.: Conformance checking based on partially ordered event data. In: F. Fournier, J. Mendling (eds.) Business Process Management Workshops - BPM 2014, Revised Papers, *LNBIP*, vol. 202, pp. 75–88. Springer (2014)
27. Mannhardt, F., de Leoni, M., Reijers, H.A., van der Aalst, W.M.P.: Balanced multi-perspective checking of process conformance. Computing **98**(4), 407–437 (2016). DOI 10.1007/s00607-015-0441-1. URL `http://dx.doi.org/10.1007/s00607-015-0441-1`
28. Meyer, A., Weske, M.: Weak conformance between process models and synchronized object life cycles. In: X. Franch, A.K. Ghose, G.A. Lewis, S. Bhiri (eds.) ICSOC 2014, *LNCS*, vol. 8831, pp. 359–367. Springer (2014). DOI 10.1007/978-3-662-45391-9_25. URL `https://doi.org/10.1007/978-3-662-45391-9_25`
29. Munoz-Gama, J.: Conformance Checking and Diagnosis in Process Mining - Comparing Observed and Modeled Processes, *LNBIP*, vol. 270. Springer (2016)
30. Munoz-Gama, J., Carmona, J., van der Aalst, W.M.P.: Single-entry single-exit decomposed conformance checking. Inf. Syst. **46**, 102–122 (2014)
31. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4), 541–580 (1989)
32. Olivé, A.: Conceptual Modeling of Information Systems. Springer, Berlin (2007)
33. OMG: Object Constraint Language - version 2.4 (2014). Available at: `http://www.omg.org/spec/OCL/2.4/PDF`
34. Popova, V., Fahland, D., Dumas, M.: Artifact lifecycle discovery. Int. J. Cooperative Inf. Syst. **24**(1) (2015)
35. Queralt, A., Teniente, E.: Reasoning on UML conceptual schemas with operations. In: CAiSE, pp. 47–62 (2009)
36. Rozinat, A., van der Aalst, W.M.P.: Conformance checking of processes based on monitoring real behavior. Inf. Syst. **33**(1), 64–95 (2008)
37. Solomakhin, D., Montali, M., Tessaris, S., Masellis, R.D.: Verification of artifact-centric systems: Decidability and modeling issues. In: Basu et al. [7], pp. 252–266
38. Störrle, H.: Semantics of control-flow in UML 2.0 activities. In: VL/HCC, pp. 235–242. IEEE Computer Society (2004)
39. Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer, Berlin Heidelberg (2007)