Embedding Reactive Behaviour into Artifact-Centric Business Process Models

Xavier Oriol[a], Giuseppe De Giacomo[b], Montserrat Estañol[a,c], Ernest Teniente[a]

[a] *Universitat Politècnica de Catalunya, Barcelona, Spain*
[b] *Sapienza Università di Roma, Rome, Italy*
[c] *Barcelona Supercomputing Center, Barcelona, Spain*

Abstract

In artifact-centric business process models it is usually assumed that the specification of the activities requires stating all the effects of the activity execution over the information base (i.e. over the artifacts it handles). In particular, these effects have to deal with integrity constraint enforcement to ensure a proper treatment of integrity constraints during activity execution. Manually specifying this treatment is a difficult, expensive and error-prone task, because of the inherent difficulty of getting rid of all the implication entailed by the constraints and also of the way to properly handle it.

In this paper, we advocate for separating constraint handling from the specification of activities in such a way that only the effects of the activity over the artifacts have to be defined (without needing to care about the constraints). Then, we propose an approach to automatically generate an extension to the original business process model that allows identifying at run-time the additional updates that have to be applied to the information base to repair all constraint violations caused by the activity execution.

Keywords: artifact-centric business process modeling, constraint repair, BPMN, UML, OCL, business process execution

Email addresses: `xoriol@essi.upc.edu` (Xavier Oriol), `degiacomo@dis.uniroma1.it` (Giuseppe De Giacomo), `estanyol@essi.upc.edu` (Montserrat Estañol), `teniente@essi.upc.edu` (Ernest Teniente)

1. Introduction

Information and processes are the two main assets of any organization [1, 2]. Information is related to the data defined through the artifacts managed by the business, together with the constraints that impose conditions on this data and which are directly drawn from the requirements of the domain of the organization. Processes correspond to the services offered by the organization to perform its business, together with the associations which establish restrictions over the order of execution of these services.

Recently, artifact-centric business process modeling, which advocates a sort of middle ground between a conceptual formalization of dynamic systems and their actual implementation, has been recognized as an appropriate approach to specify the business of an organization since it allows specifying data, processes and the link among them; and because it has shown to be quite effective in practice [3, 4, 5, 6].

Despite the variety of existing proposals to specify artifact-centric Business Process Models (BPMs), there is a large consensus that any of them must contain a conceptual model for data, such as a UML class diagram [7], which always includes a set of *integrity constraints*; and a model for the processes, expressed, e.g. in BPMN [8, 9]. Then, several alternatives exist regarding the way to establish the link between data and processes. However, this is usually achieved through the formal specification of the effects that the execution of an activity causes in the contents of the information base.

Several proposals assume that process activities are specified through OCL operation contracts [10]. Thus, for instance, linking data and processes in this way has shown to be a feasible and practical way to achieve automatic executability of artifact-centric BPMs [11]. Other languages might be chosen to establish the link, but the crucial point here is to choose a language whose expressiveness is, essentially, first-order logics (i.e., relational algebra), as it happens with the OCL expressions mostly used [12]. The approach we present in this paper is independent of the language used to specify process activities although we use OCL in our examples.

Little has been said and analyzed regarding how the specification of process activities should handle the integrity constraints in the data model. Until now, the usual approach is to assume that the specification of the behaviour of the process activities should ensure that no integrity constraint is violated after its execution (or, otherwise, the activity should be rolled back).

However, the business process and the data model can evolve independently one from the other. Therefore, in the current approach, changes in the requirements leading over the data may require modifying the business process model, at least with regards to the specification of process activities, without the business of the organization having suffered any variation. Moreover, trying to state, at design time, the intended runtime behaviour of an activity not to violate any integrity constraint is not only difficult and error-prone, but even impossible in certain situations. For this reason, best practices for requirements specification suggest that this is not an appropriate approach for the sake of facilitating requirements definition, modifiability and consistency [13].

We propose in this paper a novel approach aimed at automatically handling integrity constraints. In our approach, the definition of the process activity has to incorporate only the intrinsic changes over the data required by the business, while dealing with the constraints is left out at execution time through an automatic repairing mechanism. That makes business process definition much easier and allows the process and the data models to evolve independently.

Since constraints can be repaired in several ways, the domain expert (i.e. the person executing the process) should be allowed to choose at execution time the most appropriate action to apply in each situation. Note that the chosen repair can lead to another violation which, in turn, requires additional repairing. Selecting repairs blindly can easily lead to a wrong decision and should be avoided.

To properly deal with this phenomenon, we realized that the sequence of actions required to repair a constraint can be seen as a process. Then, all potential sequences of repairing actions may be modeled as a BPM itself. Therefore, given a constraint violation, we build a BPM-like model that shows all possible ways to repair it. Then, the domain expert may use this extended model to select the proper repairing actions by having a global sense of all the repair implications. By inspecting the model, the domain expert can see which is the shortest path to reach consistency, which is the way to avoid a certain undesired repairing action, etc., and choose the repair(s) accordingly.

Given an artifact-centric BPM, where the data is described through a data model containing integrity constraints and the behavior of the activities is described in terms of modifications over this data model, we can automatically compute, at design time, the whole chain of activities that, when executed, will repair constraint violations. Therefore, we can extend the original BPM model by considering the flow of additional activities that have to be performed to preserve integrity constraints. This extension can be computed for each activity of the original BPM. Since the computation of the extended BPM model is performed at design time, it does not negatively impact the performance of the original process execution.

Moreover, by modelling the repairing process as a kind of BPM, the process designer may customize these models at design time to forbid some undesired repairing paths or certain updates. Then, the user may use the resulting BPM at execution time to determine how to deal with constraint violations. When the execution of the extended model is finished, the execution of the business process will continue as specified in the original BPM.

The work proposed here grows from an initial proposal we presented in [14], where we outlined the technicalities regarding how to achieve this automatic behaviour. In this paper, we extend the technical contribution by providing an in-depth explanation of the treatment of the logics behind our approach, a discussion about the execution termination for the generated BPMN models, and a simplification of the generated BPMN model through activity merging. Moreover, we provide a different perspective of our work related to the use of our approach in practice and the advantages it provides to the organizations.

The remainder of the paper is structured as follows. Section 2 motivates our approach and introduces our running example. Section 3 defines basic concepts. Sections 4 and 5 explain in detail the technicalities of our contribution. Finally, sections 6 and 7 deal with the related work and the conclusions, respectively.

2. Motivation

We will motivate the need to separate the management of the business process from the treatment of integrity constraints using the following example. Assume a business process to decide whether an assistant professor with a temporal contract should be hired permanently or fired. This is a regular process we may encounter in different universities. The BPM diagram in Figure 1 states the typical activities performed to hire an assistant professor, and the order in which they should be executed. Almost identical BPM diagrams would be used by other universities pursuing the same goal.

Note that the process starts when an assistant professor is hired. Then, the time event states that a certain period of time after that hiring he has to

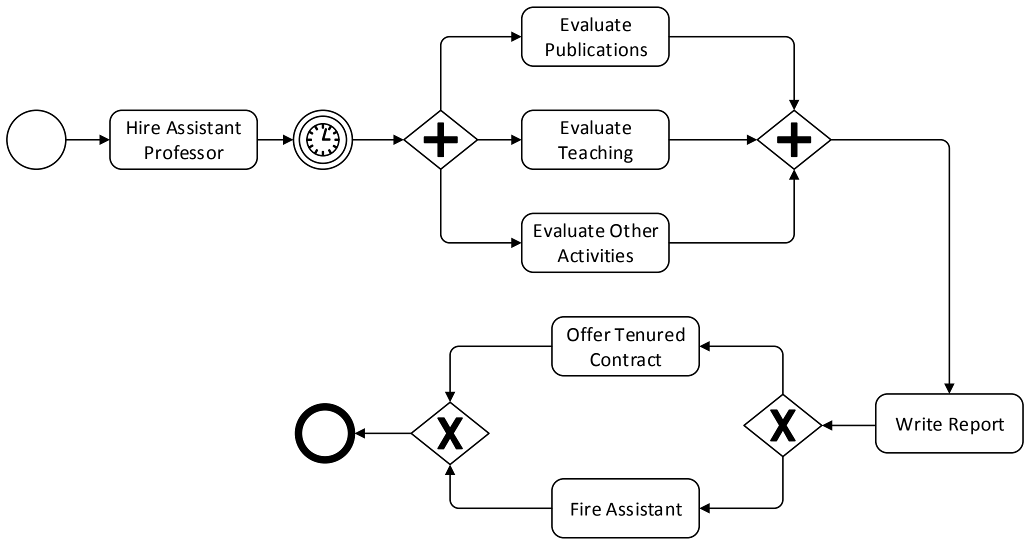

Figure 1: BPMN diagram for permanently hiring or firing an assistant professor

be evaluated for tenure. In particular, three evaluations are performed in parallel regarding his publications, his teaching and other activities performed while in their role of assistant professor. Once this is done, the commission writes a report justifying the decision taken and then the assistant is either promoted to a tenure or fired.

Making this decision requires having some data about the assistant, both regarding his activity (i.e. publications, teaching, faculty management, etc.) and work situation. Figure 2 specifies the fragment of this data regarding work situation. Note that the system stores information about current and former professors, and for current professors it states whether they are assistant or tenured. Moreover, assistant professors must have a supervisor which has to be a current professor.

The information regarding the activity of a current or former professor would be associated to the *Professor* class so that it is not lost when the professor ceases his activity at the University. For the sake of simplicity, this information is not shown in the diagram because it does not change when a professor is promoted or fired. We also omit the attributes in the figure because they are not relevant for our discussion.

Note that the class diagram contains several graphical constraints stating conditions that each state of the information base should satisfy. Thus, for

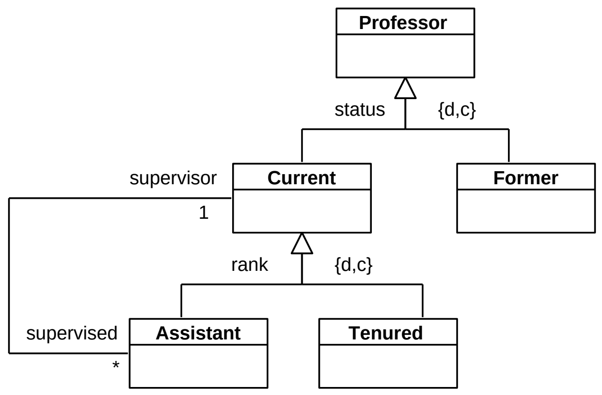

Figure 2: Fragment of the class diagram stating work situation

instance, the *Current* and *Former* classes are disjoint, i.e. a professor cannot be both at the same time, and all professors are either current or former; as stated by the *disjoint/complete* constraint at the top of the hierarchy. The same applies to assistant and tenured professors. Moreover the multiplicity constraint at the supervisor end states that an assistant is supervised by exactly one current professor. These constraints respond to specific business requirements of each university and may vary from one to another but are independent of the BPM which might the same for all universities.

We also have to specify an operation contract for each activity in the BPMN diagram to define the link between the BPM and the class diagram. We will concentrate here on the behaviour of the activities *OfferTenuredContract* and *FireAssistant*.

OfferTenuredContract does not pose any particular problem as far as the treatment of constraints goes, because switching the professor from *Assistant* to *Tenured* (and deleting the information regarding his supervisor) is enough to achieve the goal and it will never violate any constraint. So, this behaviour can be defined by means of the following OCL contract:

```
OfferTenuredContract(a: Assistant)
post: a.oclIsTypeOf(Tenured) and not a.oclIsTypeOf(Assistant)
```

However, things become more complex when having to specify *FireAssistant*. Here, switching the professor from *Assistant* to *Former* (and deleting the information regarding his supervisor) is enough to achieve the business goal. However, a constraint will always be violated if applying only this update to the information base. Note that because of this firing and given that the fired assistant may act as a supervisor of other assistants (this is not forbidden in our data model), we should ensure that all these assistants are assigned to a new a supervisor to satisfy the minimum 1 multiplicity constraint.

Assuming that the business process designer would realize of this situation (which is not trivial at all), he would start then the hard task of having to specify at design time which is the appropriate way to handle this potential violation during process execution. In particular, it is impossible for him to know which will be the most appropriate professors during process execution to substitute the fired one or the conditions under which they will be selected. Note also that this decision of having to look for new supervisors has nothing to do with the business process itself which remains valid.

One possible way to solve this problem would be to assume that a single current supervisor will take care of all substitutions. Then, we could specify it through the following contract. However, it is clear that this is not necessarily a good solution to apply whenever someone is fired, but it is impossible to do something better given the rigidity of current proposals for process activity specification.

```
FireAssistant(a: Assistant, c: Current)
post: a.oclIsTypeOf(Former) and not a.oclIsTypeOf(Current) and
    a.supervised@pre->forAll(s | s.supervisor = c)
```

This way of understanding the operation contracts corresponds to a *strict interpretation* [15]. A strict interpretation assumes passive behavior of operations, since it prevents an operation from being applied if an integrity constraint is violated (although both its preconditions and postconditions are satisfied).

Things become even more difficult within this approach when requirements evolution is taken into account. Assume, for instance, that after some years it is decided that all tenured professors should supervise an assistant. This constraint could be specified in OCL as follows:

```
context Tenured inv allSupervising:
self.supervised -> notEmpty()
```

This evolution is not related to the business process of promoting a professor, which remains the same, but to the changing conditions that the university imposes over its professors. Therefore, no changes should be applied to the BPM. However, current approaches require changing the specification of *OfferTenuredContract* because in addition to changing the assistant to tenured it is required to assign him now an assistant to supervise. Again, as before, the problem relies on deciding who this assistant will be at design time.

Further evolution may then require that all supervisors should be tenured, as stated by the following OCL constraint. This would imply additional changes, again difficult to identify and to specify, on both *OfferTenuredContract* and *FireAssistant*.

```
context Assistant inv supervisorIsTenured:
self.supervisor.oclIsTypeOf(Tenure)
```

We have seen so far the strong drawbacks of having to incorporate the treatment of constraints into the specification of the activities. However, we can overcome them by considering an *extended* interpretation of operation contracts [15] and delaying the treatment of constraints at execution time as we propose to do through the techniques proposed in this paper.

An extended interpretation of an operation assumes that the operation, when executed, not only applies the specified behavior in its contract but also all the necessary changes to ensure that no constraint is violated. That is, the operation entails some repairing reactive behavior. In this way, there is no need to specify additional effects in the postcondition to deal with constraints. As a consequence, the evolution of the requirements over the data model will not affect at all the BPM and the definition of its activities.

According to this proposal, the following simple OCL contracts would be enough to specify initially the behaviour of *OfferTenuredContract* and *FireAssistant* and should not be modified because of evolution.

```
OfferTenuredContract(a: Assistant)
post: a.oclIsTypeOf(Tenured) and not a.oclIsTypeOf(Assistant)
```

```
FireAssistant(a: Assistant)
post: a.oclIsTypeOf(Former) and not a.oclIsTypeOf(Assistant)
```

Then, the approach we propose in this paper would suffice to properly handle at execution time possible integrity constraint violations entailed by the execution of these activities. In this way we make BPM specification simpler and more appropriate to the actual behaviour of the organizations.

3. Basic Concepts

In this section, we give an overview of the logic background and notation used throughout the paper.

Terms, atoms and literals A *term* t is either a variable or a constant. An *atom* is formed by a n-ary *predicate* p together with n terms, i.e., $p(t_1, ..., t_n)$. We may write $p(\bar{t})$ for short. If all the terms $\bar{t}$ of an atom are constants, we say that the atom is *ground.* A literal l is either an atom $p(\bar{t})$, a negated atom $\neg p(\bar{t})$, or a built-in literal $t_i\ \omega\ t_j$, where ω is an arithmetic comparison (i.e., $<, \leq, =, \neq$).

Derived/base predicates A predicate p is said to be *derived* if the boolean evaluation of an atom $p(\bar{t})$ depends on one or more derivation rules, otherwise, it is said to be *base.* A *derivation rule* is a rule of the form:

$$\forall \bar{t}.\ p(\overline{t_h}) \leftarrow \phi(\bar{t})$$

Where $\overline{t_h} \subseteq \bar{t}$. In the formula, $p(\overline{t_h})$ is an atom called the *head* of the rule and $\phi(\bar{t})$ is a conjunction of literals called the *body.* We assume all derivation rules to be safe (i.e., all the variables appearing in the head or in a negated or built-in literal of the body also appear in a positive literal of the body) and non-recursive. Given several derivation rules with predicate p in its head, $p(\bar{t})$ is evaluated to true if and only if one of the bodies of such derivation rules is evaluated to true.

We extend the notion of base/derived predicates to atoms and literals. That is, when the predicate of some atom/literal is base, we say that such atom/literal is base too, otherwise, we say that it is derived.

Instance, and instantiation A ground atom of some base predicate p is called an *instance* of p. Then, a finite set I of instances of one or more predicates is called an instantiation.

Substitution A *substitution* σ is a set of the form $\{x_1/t_1, ..., x_n/t_n\}$ where each variable x_i is unique. The domain of a substitution is the set of all x_i and is referred as $dom(\sigma)$. We say that σ is ground if every t_i is a constant. The literal l_σ is the literal resulting from simultaneously substituting any occurrence of x_i in l for its corresponding t_i. Similarly, we define the conjunction ϕ_σ as the conjunction resulting from simultaneously applying the substitution σ to all the literals of ϕ.

Denial constraints A *denial constraint* is a rule of the form:

$$\forall \bar{t}.\ \phi(\bar{t}) \rightarrow \bot$$

Where ϕ is a conjunction of (possibly derived) literals and $\perp$ is an atom that evaluates to false. We suppose all denial constraints to be safe (i.e., each variable appearing in a negated or built-in literal also appears in a positive literal). Intuitively, the left hand side (LHS) of a denial constraint express a condition that should never be satisfied by an instantiation.

Disjunctive embedded dependencies A *disjunctive embedded dependency* (*ded*) is a rule of the form:

$$\forall \overline{t}.\ \phi(\overline{t_\phi}) \rightarrow \bigvee_{i=1..n} \exists \overline{y_i}.\ \psi_i(\overline{t_i}, \overline{y_i})$$

where all literals are positive and base. It is important to highlight that n might be 0, and thus, the right-hand side might be empty. In such case, we use the convention that the empty disjunction evaluates to false [16] and write $\perp$ to represent so. Note that *deds* are a kind of tuple-generating dependencies allowing disjunctions in the right hand side.

4. Generating violation handling extensions in BPM

We describe in this section our approach for automatically embedding the reactive behaviour into the original BPMN model specified by the business designer. We provide first a general overview of the approach, and then explain in detail the six steps that have to be performed. Finally, we provide a discussion about the practicality of our approach in a real life setting.

4.1. Overview

When executing any process activity, a violation of a constraint can occur. As we have seen, this violation can be repaired by considering additional updates to perform. However, this may in turn violate other constraints again, thus, forcing the execution of more updates to preserve the consistency of the information base. This is the inherent difficulty of the problem of integrity constraint repairing.

Fortunately, the constraints that might be violated when repairing other constraints can be determined at design time; i.e., we can identify them by inspecting the constraints' definition itself, without considering the contents of the information base. Indeed, several approaches build a dependency-graph showing this relation among the constraints [17, 18]. So, the idea is that, to repair a constraint violation C and to ensure that no other constraint has been violated, we have to repair C, check the constraints pointed out by

C and repair them if necessary (which might require inspecting and repairing other constraints, recursively).

In essence, our idea is that we can see the dependency graph as a BPMN diagram establishing which activities have to be carried out (and in which order) to repair a constraint violation. That is, each activity in the diagram stands for an update to apply in order to repair a constraint violation. Then, this activity is followed by those additional activities that repair the constraint that might have been violated because of the previously applied data update. When we reach the final BPM end event, we are sure that the initially violated constraint has been repaired, and that it has been repaired in such a way that no other constraint is now violated.

More in detail, our method uses the following steps which will be further explained in the remainder of this section:

1. *Translating integrity constraints into RGDs.* Repair-generating dependencies (RGDs) are logic formulas that, given an information base state and a data update, derive new updates that must be applied to repair a constraint violation [19]. In this step, we translate the constraints into the corresponding RGDs.
2. *Building the dependency-graph of RGDs.* When executing RGDs to derive new updates, one RGD can cause the violation of another constraint, thus triggering the execution of another RGD. In the dependency-graph, we explicitly show this interaction, i.e., which RGDs might trigger other RGDs.
3. *Associating each activity to the affected part of the dependency-graph.* Given an activity in the initial BPMN, its execution can only violate some constraints, thus triggering only some specific RGDs from the dependency-graph. In this step, we automatically prune all those RGDs that can never be triggered.
4. *Translating the dependency-graph fragment into a BPMN diagram.* Intuitively, RGDs are translated as activities in the BPMN diagram and the dependency-graph edges determine the flow between them.
5. *Merging activities.* Given the activities, it is possible to merge some of them in order to reduce the size of the BPMN diagram. The rationale behind merging activities is that some of them are always executed consecutively, which means that they can be merged into a single one that executes the overall effect.

6. *Customization.* Finally, the BPM designer can decide to prune some of the suggested ways to repair a constraint in the BPMN diagram. Indeed, our method generates all possible activities that can be applied to repair a violation. However, it may be the case that some of them are not desirable in the domain. In this step, we show how to prune at design time the undesired repair actions.

In this way, for each activity in the initial BPMN model, we compute its BPM extension which guarantees that, when executed, it checks and repairs all violations that may occur. This extension could then be integrated in the original BPM through a CASE tool, and be used at run time to repair constraint violations through a process executor, such as [11]. The visualization of the computed BPMN model extension is out of the scope of this paper which concentrates on how to automatically obtain and execute this extension.

It is worth mentioning also that, although we use BPMN and UML/OCL in our examples, other notations, like *service blueprints* for instance, might be used as well as long as they are detailed enough to be executed [20]. In particular, we only need these notations to be translatable into first-order logics, which is the basic framework of our approach.

It is well-known that a BPM instance execution is not transactional since it is usually implemented as a sequence of database transactions. Thus, the state of the world may change due to concurrent process execution during a repair execution introducing more inconsistencies or an activity execution may fail. Nevertheless, in this paper we assume that two tasks cannot be executed simultaneously, as they might interact to cause a constraint violation, and this is one limitation of our approach. Therefore, in such cases, these tasks should be serialized. An approach for detecting non-parallelizable activities due to constraint conflicts can be found at [21], but an in-depth analysis of concurrency is left out for further work.

4.2. Translating UML/OCL constraints into RGDs

RGDs are tuple-generating dependencies that, given an information base state and a set of updates, derive new updates that perform a repair of a constraint violation [19]. RGDs can be automatically obtained from the UML/OCL constraints in the conceptual schema by performing the following three steps:

1. Translating UML/OCL constraints into denials

2. Incorporating events into denials
3. Transform event rules into RGDs

Note that, although we only talk about integrity constraints in this paper, the UML/OCL languages allow for the definition of structural constraints regarding the information model but also semantic constraints, such as the business rules required by the organization. Some examples of these semantic constraints have been shown in our motivating example. The advantage of our approach is that we can treat all of them uniformly.

4.2.1. Translating UML/OCL constraints into denials

A *denial* is a rule stating that a certain condition (as given by the expression in its body) can never hold in an information base state. I.e. they define situations that can never happen as well as integrity constraints do. Denials are specified as logic rules with $\perp$ in its head.

UML/OCL constraints, i.e. textual constraints defined in OCL and graphical and implicit constraints in the UML schema, can be automatically translated into denials as given in [17]. The rationale behind is that most UML and OCL constraints are equivalent to first order logics [12], and every first-order constraint can be rewritten as a denial [22].

As an example, consider the UML specialization constraint stating that each *Assistant* is a *Current* professor, and the implicit referential constraint stating that each *Supervisor* is also a *Current* professor. They can be translated to the following denials:

$$Assistant(x) \land \neg Current(x) \rightarrow \perp$$
$$Supervises(x, y) \land \neg Current(x) \rightarrow \perp$$

Intuitively, the first rule states that, if there is an *Assistant* x, but x does not appear as a *Current* professor, then there is a constraint violation. Similarly, the second rule specifies that if x supervises y, but x is not a *Current* professor, then, there is also a constraint violation.

We assume in this paper that constraints in the conceptual schema are specified by means of the UML/OCLuniv subset [23]. This entails two main advantages: 1) it ensures that the generated process extensions will always terminate despite the loops that can appear in the BPM (as we will show in Section 5.2); 2) obtaining RGDs is much simpler for UML/OCLuniv than for general UML/OCL. It is worth mentioning, however, that denials can be obtained from any first-order equivalent language.

We consider that all UML/OCL constraints are translated into denials as given in [17] except for minimum multiplicity constraints whose rewriting into RGDs will be made adhoc as explained later.

4.2.2. Incorporating events into denials

Denials only refer to the contents of the information base, but they do not take into account the update that can lead to the violation of the condition they define. Therefore, they do not provide enough information to embed in the BPM the required reactive behaviour. We need to incorporate events, i.e. updates, into the denials with this purpose.

An update of the information base may be either an insertion, denoted by the event $\iota P(\overline{a})$ or a deletion, denoted by $\delta P(\overline{a})$. Given a fact $P(\overline{a})$ in the current information base and the events in the update, we can deduce whether $P(\overline{a})$ will hold in the new, updated, information base according to the following event rule equivalences [24]:

$$P^N(\overline{x}) \equiv \iota P(\overline{x}) \vee P(\overline{x}) \wedge \neg\delta P(\overline{x})$$
$$\neg P^N(\overline{x}) \equiv \delta P(\overline{x}) \vee \neg P(\overline{x}) \wedge \neg\iota P(\overline{x})$$

The first rule states that a fact $P(\overline{x})$ will be true in the new state (denoted by $P^N(\overline{x})$) if and only if an insertion event $\iota P(\overline{x})$ happens in the update or if $P(\overline{x})$ was already true in the old state and it has not been deleted ($P(\overline{x}) \wedge \neg\delta P(\overline{x})$). The rule for $\neg P^N(\overline{x})$ works similarly.

Event rule equivalences are sound and complete to define the truth value of a fact after the application of an update. Sound in the sense that the changes they define are correct and complete because no other rule is needed to define all possible changes. They can be understood as a sort of "frame axioms" to specify that all facts not affected by the events are not changed while executing that update [25].

We can incorporate events into denials by replacing each literal in the initial denial by the previous event rule equivalences and then transforming the result into conjunctive normal form. In this way we will obtain several rules with events for each denial. They correspond to the different ways an update can violate the condition stated by the denial.

When we apply the replacement given by the equivalence event rules to the denials in our running example we obtain:

$$\begin{aligned}
&\iota Assistant(x) \wedge \neg Current(x) \wedge \neg \iota Current(x) \to \bot\\
&Assistant(x) \wedge \neg \delta Assistant(x) \wedge \delta Current(x) \to \bot\\
&\iota Assistant(x) \wedge \delta Current(x) \to \bot
\end{aligned}$$

$$\begin{aligned}
&\iota Supervises(x,y) \wedge \neg Current(x) \wedge \neg \iota Current(x) \to \bot\\
&Supervises(x,y) \wedge \neg \delta Supervises(x,y) \wedge \delta Current(x) \to \bot\\
&\iota Supervises(x,y) \wedge \delta Current(x) \to \bot
\end{aligned}$$

The first three rules ensures that an update will not violate the constraint that nobody can be an assistant but not a current professor. The first rule states that if x has been inserted as an assistant, but he was not current before nor inserted as such, then the constraint is violated. The second one prevents an assistant to be deleted as current but not as an assistant. The third one specifies that it is not possible to insert an assistant but delete him as current in an update. Rules for the second constraint behave in a similar way.

4.2.3. From event rules to RGDs

Once we have incorporated events into the denials, the RGDs for each denial can be easily obtained moving the negated events from the left hand side (LHS) of the rule to its right hand side (RHS). This corresponds to following the logical equivalence $A \wedge \neg p \to C \equiv A \to C \vee p$.

Applying this transformation to the rules in our running example we obtain the following RGDs:

$$\iota Assistant(x) \wedge \neg Current(x) \to \iota Current(x) \tag{1}$$
$$Assistant(x) \wedge \delta Current(x) \to \delta Assistant(x) \tag{2}$$
$$\iota Assistant(x) \wedge \delta Current(x) \to \bot \tag{3}$$

$$\iota Supervises(x,y) \wedge \neg Current(x) \to \iota Current(x) \tag{4}$$
$$Supervises(x,y) \wedge \delta Current(x) \to \delta Supervises(x,y) \tag{5}$$
$$\iota Supervises(x,y) \wedge \delta Current(x) \to \bot \tag{6}$$

RGD 1 states that if a new *Assistant x* is inserted when *x is not a Current professor*, then it must also be inserted that *x is a Current* professor to ensure that the information base state does not violate the constraint that nobody can be an assistant but not to be a current professor. RGD 2 behaves in a similar way by repairing the constraint through a deletion of x as an *Assistant*.

RGD 3 is not properly a repair-generating dependency since it specifies a situation where no additional event can be applied to satisfy the constraint. This is so because the events in the update are contradictory and cannot be applied together. In this case, we have that it is not possible to insert someone as an assistant and delete him as a current professor at the same time. In general, any RGD with $\perp$ in the head cannot be repaired.

RGDs 4, 5 and 6 behave in a very similar way.

It is worth noting that not all RGDs are deterministic since some violations can be repaired in different ways. RGDs capture this non-determinism by means of disjuncts and existential variables in the RHS.

Disjunctions in the RGDs will appear when moving several negated events from the LHS to the RHS of a rule. For instance, obtaining the RGDs for the completeness constraint that states that each *Current* professor is either *Assistant* or *Tenured* will give raise to the following RGD with disjunctions in its head:

$$\delta Assistant(x) \land \neg Tenured(x) \land Current(x) \rightarrow \delta Current(x) \lor \iota Tenured(x) \tag{7}$$

Intuitively, this RGD states that if we delete an assistant professor, we either also delete him as current professor or insert him as tenured. As before, this RGD would be obtained by translating first the constraint into a denial, incorporating events into the denial and moving negated event literal in the LHS to its RHS.

All UML/OCLuniv constraints are translated to RGDs as we have explained so far. The only exception are the minimum multiplicity constraints 1 in the conceptual schema. In this case we propose making a direct translation to RGDs, as we did in [19].

As an example, the multiplicity constraint stating that each assistant is supervised by one current professor would give raise to the following RGDs:

$$\iota Assistant(x) \rightarrow \iota Supervises(y, x) \tag{8}$$

$$\delta Supervises(y, x) \land \neg OtherSuper(x) \rightarrow \delta Assistant(x) \lor \iota Supervises(z, x) \tag{9}$$

$$OtherSuper(x) \leftarrow Supervises(z, x) \land \neg\delta Supervises(z, x)$$

The first rule states that, when inserting a new assistant professor x, we should also insert a supervisor y for x. The second RGD detects a violation if we delete a supervisor y of an assistant professor x, and x does not have any other supervisor. This is detected through the *OtherSuper* derivation rule (defined below rule 9)[1], which specifies that x will have another supervisor if it has a supervisor z that is not being deleted.

Note that RGD 9 is non-deterministic since it requires choosing between deleting the assistant professor x, or adding a new supervisor z for him/her. The second choice is non-deterministic since we can chose different values for z (i.e. its value is not bounded by the LHS of the rule).

Given a general minimum multiplicity constraint 1 for member M_2 of an association A between M_1 and M_2, we will always obtain non-deterministic RGDs (and a derivation rule) defined by the following pattern:

$$\iota M_1(m_1) \rightarrow \iota A(m_1, m_2)$$
$$\delta A(m_1, m_2) \wedge \neg \mathit{OtherM_1}(m_1) \rightarrow \delta M_1(m_1) \vee \iota A(m_1, m_2')$$
$$\mathit{OtherM}_1(m_1) \leftarrow A(m_1, m_2') \wedge \neg \delta A(m_1, m_2')$$

It is worth noting that in our approach existential variables will only appear in an RGD when translating UML minimum multiplicities.

Given the domain where the business process has been defined, it is likely that some of the events will never happen. For instance, *Professor* objects are never deleted in our example since we keep a historical track of professorships (thus moving them from current to former when they leave). For this reason, some of the RGDs we obtain can be simplified (by removing the literals corresponding to those events that will never happen) or even removed if they only refer to all such events.

The events that do not happen in a domain can be drawn from the UML class diagram itself. When a class/association A is *add-only*, the event of deleting an instance of A cannot take place. Similarly, if a class/association A is *frozen*, no insertion nor deletion event on A can be in the update. In our example, *Professor* is an add-only class.

4.3. Building the dependency-graph of RGDs

Given a set of RGDs, we can build a dependency-graph which shows the RGDs that may trigger the execution of other RGDs. For instance, assume that a process removes an assistant professor x. RGD 7 states that we have to additionally choose between inserting x as tenured or deleting x as current. If we opt for deleting x as current professor, we can trigger all RGDs having

[1]Don't confuse *RGDs* (generating new tuples to be inserted/deleted, denoted by $\rightarrow$), with *derivation rules* (deducing contents of the information base, denoted by $\leftarrow$).

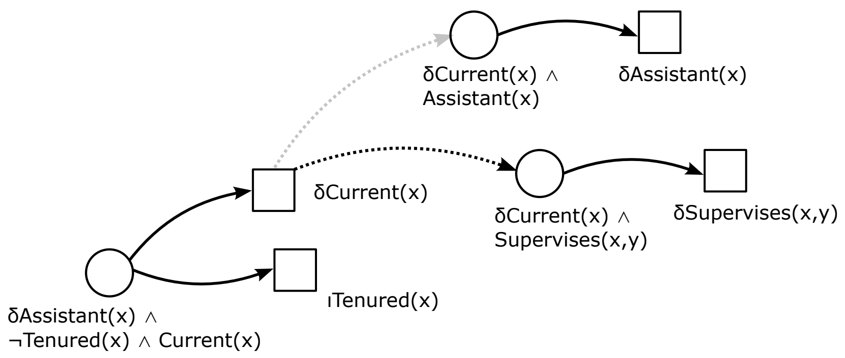
δCurrent(x) ∧ Assistant(x)
δAssistant(x)
δCurrent(x)
δCurrent(x) ∧ Supervises(x,y)
δSupervises(x,y)
ιTenured(x)
δAssistant(x) ∧ ¬Tenured(x) ∧ Current(x)

Figure 3: Dependency graph for RGDs 7, 2, and 5

δCurrent in its LHS. This happens with RGDs 2 and 5, for instance. Then, we have that RGD 7 *can trigger* RGDs 2 and 5.

For the sake of self-completeness, we summarize in the following how to build a dependency-graph from a set of RGDs as proposed in [18].

The dependency-graph contains a round vertex (called *constraint-vertex*) for each LHS of a RGD, and also a square vertex for each structural event in the RHS of a RGD (called *repair-vertex*). There is an edge (straight arrow) from the constraint-vertex of an RGD to each one of its repair-vertices to state that if the condition in the constraint-vertex is satisfied, one of its repair-vertices must be executed. Moreover, there is an edge (dotted arrow) from a repair-vertex to each of the constraint-vertices that may have been violated because of the execution of the repair.

As an example, the triggering relationship between RGDs 7 and RGDs 2 and 5 are depicted in Figure 3.

In general, there is a triggering relationship between the repair vertex R of a RGD to the constraint vertex C of another RGD if R and C have an event in common. Indeed, this means that the repair of the first constraint is an update that can potentially violate the second constraint.

Some of the edges in the dependency-graph can be safely removed, as stated in [17, 18], since they will never be triggered. For instance, the edge between RGD 7 and RGD 2 will never be applied since if RGD 7 is fired, then RGD 2 can never be violated. This is so because when violating the completeness constraint by deleting an assistant professor, and repairing it by deleting him as current, it is impossible to violate the hierarchy constraint (which states that every *associate* professor is a *current* professor).

We say that a trigger between a repair-vertex R_1 from RGD_1 and a constraint-vertex C_2 from RGD_2 is lively if and only if there is an instance of the data D that satisfies RGD_1 through R_1 (i.e., $D \models LHS(RGD_1)\sigma$ and $D \models R_{1_\sigma}$ for some σ), but violates C_2 (i.e., $D \models C_2\sigma$ for some σ, but $D \not\models RHS(RGD_2)\sigma\sigma_2$ for any σ_2). Triggers that are non lively are removed from the dependency graph. They can be detected through syntactic criteria, such as finding $p \wedge \neg p$ contradictions, or realizing that some repair-vertex is subsumed by some constraint-vertex (as in the case of the completeness vs hierarchy constraint RGDs).

4.4. Associating activities to the dependency-graph

The dependency-graph we have obtained so far is built by taking only into account the definition of the integrity constraints, but this graph should also incorporate the activities in the BPMN model to be able to determine the constraints that might be violated when the BPMN activities are executed. This is achieved by specifying the BPM activity as an RGD, including this RGD in the dependency-graph, and identifying the RGDs in the former graph reachable from it.

A BPM activity can be seen as an RGD whose repair is, in fact, the execution of the update in its postcondition. For instance, consider the activity *fireAssistant*, from Figure 1, stating the removal of an assistant professor. The effect of this activity can be specified by means of the following RGD:

$$\mathit{fireAssistant}(x) \rightarrow \delta \mathit{Assistant}(x) \wedge \iota \mathit{Former}(x) \tag{10}$$

Note that the execution of an operation can lead to several events which can combine insertions and deletions freely, as it happens in our example. Moreover, we handle attribute updates using the classic encoding of an update as a deletion and an insertion of the same tuple with the new values changed. This is sound since all insertions and deletions from the repairing BPM are only applied and checked for consistency at the end of the process.

A similar way for deriving the events entailed by the execution of an activity is already used in [11], where an automatic translation from BPM activities specified through OCL constraints into this kind of rules is given [2].

Now the new RGDs are included in the dependency-graph as shown in Figure 4. In our example, the new RGD $\delta Assistant(x) \wedge \iota Former(x)$ points

[2]They are not explicitly named RGDs in [11], but the way they are formalized makes them an RGD in practice

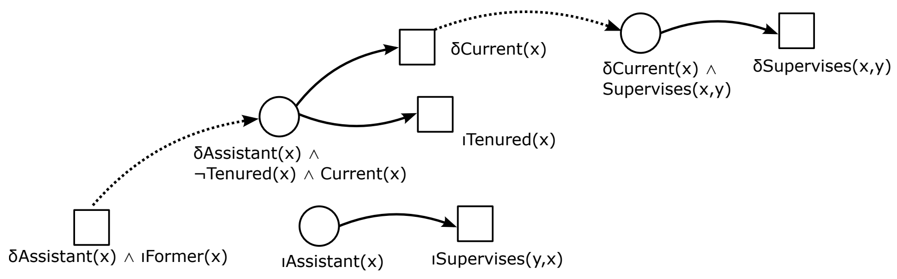
δCurrent(x)
δCurrent(x) ∧ Supervises(x,y)
δSupervises(x,y)
ıTenured(x)
δAssistant(x) ∧ ¬Tenured(x) ∧ Current(x)
δAssistant(x) ∧ ıFormer(x)
ıAssistant(x)
ıSupervises(y,x)

Figure 4: Dependency graph including an RGD representing the effect of an activity

to RGD 2 since both have the $\delta Assistant$ predicate in common. However, it deoes not point to RGD 8 since they have no shared predicate.

As a result, we have that the RGDs possibly affected by the execution of a BPM activity are those in the fragment of the dependency-graph reachable from the RGD encoding the effect of that activity. In our example, RGD 7 and RGD 5, but not RGD 8. RGDs that are not reachable from these new RGDs are removed since they refer to constraints that will never be violated during the integrity maintenance process raised from the activity.

4.5. Translating the dependency-graph into a BPMN diagram

Once we know the relevant part of the dependency-graph, we translate it into a BPMN diagram. This will allow us to pilot the process of integrity maintenance in the same way as classical BPMN diagrams. The basic idea of the translation is that constraint-vertices are translated to BPMN gateway events that allow a user to choose between the available repairs, while a repair-vertex becomes a BPMN activity that applies the repair. Then, these activities are followed either by an OR-gateway which points out to the (BPMN translation of) constraint-vertices that may have been violated because of the applied repair or by an end-event if no constraints is violated.

More precisely, the translation of a constraint-vertex depends on the number of repair-vertices it has. If there is no repair, the constraint-vertex becomes a BPMN error event meaning that if we reach the violation of that constraint then there is no possible way to repair it and an error is thrown. If there is a single repair, the constraint-vertex becomes the BPMN-activity that applies it. If there is more than one potential repair, the constraint-vertex is translated to an event-gateway that enables the domain expert choosing his preferred way to repair the violation.

The translation of a repair-vertex always produces a unique activity that applies the changes that repair the constraint. This activity may require domain expert input to choose the value for the existential variables since this entails decision making at business level. For instance, in RGD 9 we have a repair vertex which inserts a new supervisor z to an assistant x. If the domain expert selects this repair, he will be required to explicitly choose a specific value for z at execution time.

After applying a repair, several constraints can be violated. The OR-gateway will take care of this situation since several paths will be activated in this case. They will lead to the constraint-vertices that allow to perform the repair of the newly violated constraints.

We ensure that we only repair actually violated constraints through the guard conditions in the OR-gateway's outgoing flows. That is, an outgoing flow pointing to a constraint vertex c has, as guard, the logic condition encoded in c. Thus, the only way to execute an activity that repairs a violation is through the guard that first checks the constraint. So, the repairs will only be performed when the repair needs to be applied.

Note that we do not use OR-joins for synchronizing the activities execution. Such synchronization is not necessary since each path execution represents a different violation repair strategy for some particular values, and the repair for such values is independent from the rest of violations and repairs. We capture this behaviour using OR-gateways without OR-joins for ease of readability[3].

The translation from a dependency-graph to a BPMN diagram is formally given by Algorithm 5. Its input parameters are the (relevant part of) the dependency graph and the constraint-vertex representing the BPMN activity that triggers the maintenance process (behaving as start activity). As output, the algorithm provides the resulting BPMN diagram. It is easy to see that the algorithm runs in polynomial time with regards to the input.

Figure 6 shows the result of applying Algorithm 5 to the RGDs in Figure 4. The obtained BPM shows that when executing *fireAssistant* it may happen that we satisfy all the constraints, or that we need to either insert the deleted assistant as a tenure, or to delete him as current professor. If the domain

[3]If the business expert prefers avoiding this kind of diagrams, since OR-gateways are usually synchronized with OR-joins, our method can be adapted to replace OR gateways by a combination of XORs and tasks.

```
BPMN translateGraph(Dependency-graph g, ConstraintVertex startCV){
   Map<ConstraintVertex, BPMN-Node> c-map = new Map();
   Map<RepairVertex, BPMN-Activity> r-map = new Map();
   //   Creating the BPMN and adding the start/final node
   BPMN bpmn = new BPMN();
   BPMN-StartEvent start = new BPMN-StartEvent();
   BPMN-EndEvent end = new BPMN-EndEvent();
   //   Translating Constraints
   for(ConstraintVertex cv: g.getConstraintVertices()){
      Set<BPMN-Activity> repairingActivities = new Set();
      for(RepairVertex rv: cv.getRepairVertices()){
         BPMN-Activity repairActivity = createRepairActivity(rv);
         repairingActivities.add(repairActivity);
         r-map.put(rv, repairActivity);
      }
      BPMN-Node cv-node;
      if(repairingActivities.isEmpty()) cv-node = new BPMN-ErrorEvent();
      else if(repairingActivities.size() == 1)
              cv-node = repairingActivities.pop();
      else cv-node = new BPMN-EventGateway(repairingActivities);
      c-map.put(cv, cv-node);
      bpmn.add(cv-node);
   }
   //   Adding the start
   start.addNext(c-map.get(startCV));
   //   Link repairs to Constraints
   for(RepairVertex rv: g.getRepairActivities()){
      Map<Condition, BPMN-Node> bpmn-cons = new Map();
      for(ConstraintVertex cv: rv.getNextConstraintVertices()){
         bpmn-cons.put(cv.getViolationCondition(), c-map.get(cv));
      }
      if(bpmn-cons.isEmpty()) r-map.get(rv).addNext(end)
      else {
         BPMN-Node bpmn-or = new BPMN-OrGateway(bpmn-cons);
         if(bpmn-cons.size() == 1) bpmn-or = c-map.get(bpmn-cons.get(1))
         bpmn-or.addDefault(end);
         r-map.get(rv).addNext(bpmn-or)
}}}
```

Figure 5: Algorithm for obtaining the BPMN diagram from the dependency-graph

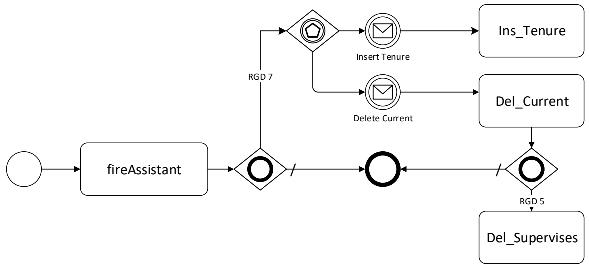

Figure 6: BPMN diagram for repairing activity *fireAssistant* if there is a violation

expert decides to delete him as current professor, he will also need to remove its supervisor relationships, if any.

4.6. Merging Activities

In the BPM obtained so far, each activity represents a single event to apply. However, BPM activities can capture in general more than one event and achieving it will also allow reducing the size of the model. Thus, we aim now at merging these single-event activities into more complex activities. We first explain the intuition behind our approach through our running example, and then, present its logic foundations.

4.6.1. Merging Activities intuition

Merging activities is devoted to reducing the size of the generated BPM. However, we have to carefully select which activities can be merged. For instance, two parallel activities following a gateway should not be merged since we would lose the information provided by the gateway. Hence, in our example, the activities *ins_ Tenure* and *del_ Current* should not be merged since we would lose the semantics of the gateway stating that the domain expert must choose between the two.

We can merge two activities when this will not alter the semantics of the resulting BPM. That is, the repairs captured by the BPM. This situation occurs when the execution of an activity determines necessarily the execution of another one, with no free choice by the domain expert. In general, any activity coming from an RGD whose RHS is composed of only one event, and that does not contain any existential variable, can be merged with the activities that might potentially violate it. This is because the absence of

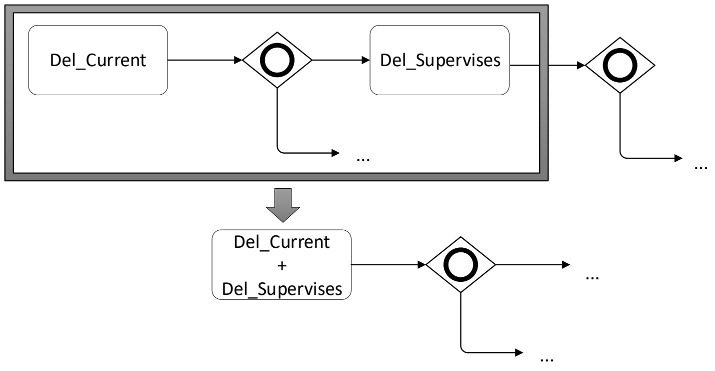

Figure 7: Example showing the merge of two activities

another repairing event and the absence of existential variables makes the execution of this extra event absolutely deterministic.

For instance, in Figure 7 we show an extended BPM for our running example, showing that, after deleting any current professor, we must delete all its supervising relationships. Note that the second activity has to be applied in case there is a violation, and that the domain expert has no free-choice. Thus, they both can be merged into a single one.

4.6.2. Foundations for merging activities

Two activities p and q can be merged if the RGD corresponding to activity q has the form:

$$\phi(x, y) \land p(x) \to q(x) \tag{11}$$

where ϕ is an arbitrary conjunctions of literals.

In the current translation into BPMN, the $p(x)$ activity leads to an OR-gateway that checks whether RGD 11 is being violated (through a guard-condition). If this is the case, the OR-gateway leads the process to a $q(x)$ activity, which in turn, can imply the violation of other RGDs. This is shown in Figure 8 (a). Note that, whenever we execute $p(x)$, we have to execute $q(x)$ if $\phi(x, y)$ is true. This is a deterministic behavior and the domain expert cannot make any choice about it.

Instead of encoding $\phi(x, y)$ as a guard-condition that leads to $q(x)$, merging allows to incorporate the *if condition* inside the activity of $p(x)$ and apply

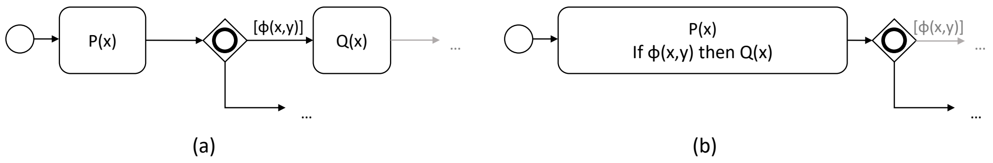

Figure 8: Showing the merge of two activities

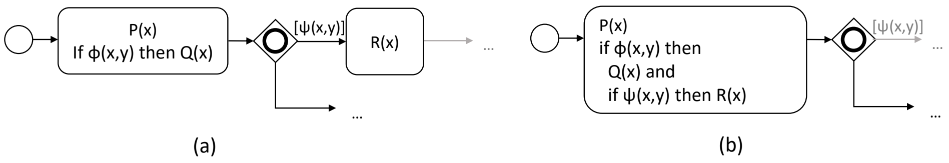

Figure 9: Showing the merge of two activities

$q(x)$ at the very same time as $p(x)$. In this way, we reduce the number of nodes from the BPMN. This is shown in Figure 8 (b).

In case there are several activities that could violate RGD 11, $q(x)$ should be merged with all of them. Note that if $q(x)$ can be merged with one activity, it can be merged with any other one that points to it. This is because the determining factor of merge-ability is the determinism of the RGD 11.

The process of merging can be recursively applied. Indeed, consider an extra RGD:

$$\psi(x, y) \wedge q(x) \rightarrow r(x) \tag{12}$$

Again, $q(x)$ might violate RGD 12, and if this is the case, we will have to apply $r(x)$ for sure. Thus, we can merge $r(x)$ activity inside the activity where $q(x)$ is placed, that is, $p(x)$. This is shown in Figure 9, where (a) represents the BPMN before merging, and (b) the result after merging.

It is worth mentioning that the process of merging we apply right now is non-deterministic. That is, given a BPMN, several possible merges can be applied. Nevertheless, we conjecture that, no matter the order, applying all the merges until no other merge can be applied will always bring the same final *fixpoint* BPMN. Checking this conjecture is left as further work.

4.7. Customization

The merged BPMN diagram represents all possible ways to repair the various constraints that can eventually be violated by the activity execution.

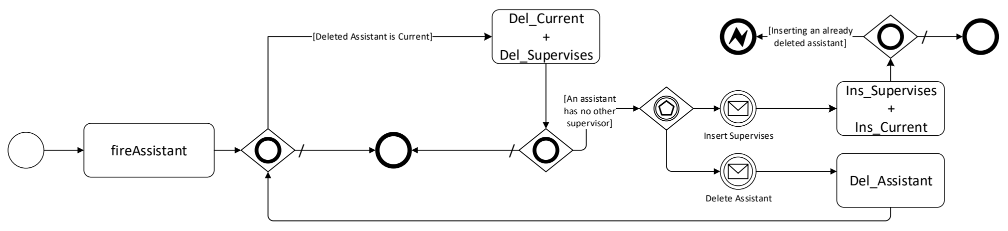

Figure 10: Final BPMN for the fireAssistant operation.

This is due to the fact that RGDs capture all possible ways to repair a constraint [19], and all the RGDs are represented in the BPMN diagram.

However, it might be the case that some of the proposed repairs are not desirable in the domain of the problem. For instance, in our running example, a domain expert might consider that, repairing the *fireAssistant* operation by means of inserting *tenures* is not a valid repair since it would necessarily imply hiring new people. Then, this kind of repair should be avoided at execution time.

To achieve this, we have to consider the RGDs which result in inserting *tenures*. These RGDs are no longer appropriate and should be deleted from the dependency graph. In terms of the BPMN diagram, this implies removing any activities that insert tenures and all the subsequent activities.

In Figure 10 we show the final BPMN generated for repairing any violation occurring when executing the *fireAssistant* operation. Note that, in this BPMN, we have merged the activities *del_ Current* with *del_ Supervises*, and the activities *ins_ Supervises* with *ins_ Emp*.

Intuitively, this BPMN diagram tells us that, when we fire an assistant, we need to delete him as a current professor and, additionally, delete any supervising association he has. When doing so, it might be the case that another assistant professor ends without having a current professor supervising him/her. In this case, we have to choose between removing this unsupervised assistant, or adding a current professor to supervise him. In case we delete the unsupervised assistant, we might need to repeat all the process again (delete the assistants supervised by him/her, etc). In case we insert a new supervisor for him/her, we have to check that we do not insert as a supervisor an assistant who has been deleted (if that was the case, the repair would entail a contradiction and an exception would be raised).

Note that our proposal relies on incorporating the BPM model obtained from the RGDs into the initial BPM model so that it entails in it the reactive behaviour required to keep constraints satisfied. This is done automatically, and requires only as input the initial BPM model; the data class diagram and its integrity constraints defined in UML/OCL; and the OCL specification of the activities in the BPM. In this way, the business analyst can still define and keep the business process and the integrity constraints separately, and use our approach to endow the repairing behavior, entailed by the constraints, into the (relevant) activities of the BPM. Then, business analyst input is only required in this final customization task in order to adjust the reactive behaviour to the particular domain of the BPM. In fact, the only interaction required from the business analyst is to remove those repairing activities that are undesired according to this domain.

4.8. Discussion

It can be argued that our approach would be very complex in a real-life setting, since: 1) for each activity a repair process model should be generated; 2) this repair process model should be verified and customized by the domain expert. Moreover, the generated repair process models have to be considered in the original process model and this may make real life process models very large, and with many variations.

Although true, this complexity is due to the intrinsic nature of the automatic repairing problem, which becomes even harder when BPM processes are taken into account. However, the current alternative of embedding explicitly constraint handling in the operation contracts shows the drawbacks stated in Section 2 and makes it inapplicable in many practical situations as the ones we outlined there. Therefore, in a real life setting, some kind of compromise should be achieved between the two alternatives.

In fact, we could for instance *hide* the extensions and show them only *on-demand* so that the user would only be aware of the complexity of the repairing when this was strictly necessary. We also understand that manually revising all the generated repairing processes might be very hard in a real scenario. However, we could apply our technique only on *crucial* activities instead of all, or *on-demand* according to the needs of the business modeler.

Also, we could take the structural events of future activities into account, and apply formal reasoning techniques, to suggest how to *customize* the current repairing-process to ensure that a repairing activity allows the remaining future activities in the BPM to be executed as well.

In summary, our approach is a way to automatically generate repairing processes for some activity given a set of constraints in an artifact-centric BPM setting. This is by itself an innovative and complex problem, and the presented approach is a first step towards solving it. In any case, this current solution is already practical for generating, at least, repairing processes for pre-selected activities. It is left as further work to discuss which activities should be pre-selected, e.g. due to their being critical to the process, to apply this analysis.

5. Executing BPM extensions to repair violations

We first explain how our generated BPM extension is executed, with special emphasis on the interpretation of OR-gateways. Then, we discuss about the termination of this automatically generated BPM extensions. We end the section by using an existing BPM executor to run our generated extension to show the feasibility of our approach.

5.1. Business process extension execution semantics

Intuitively, the BPMN language is based on token semantics [26]. Each diagram node consumes and generates tokens. Roughly, when a process begins its execution, a token is generated by its start event for each of its outgoing flows. Each activity is activated when a token reaches one of its incoming flows. When finishing its execution, the activity generates a token for each of its outgoing flows. When a token reaches an OR-gateway, all the conditions of the gateway's outgoing flows are analyzed. The gateway places a token on each outgoing flow whose condition evaluates to true. If no condition is true, then, a token is placed in the default flow. For our purposes, this intuitive token semantics suffices, but it is worth mentioning that they can be formalized by means of petri-nets [27].

The key idea of our approach is that, when running our BPM extension, each token will correspond to a different constraint violation. Since there are several constraints that can be violated simultaneously, when executing the BPM extension, there might be several tokens alive simultaneously.

The generation of these tokens is done by the OR-gateways. An OR-gateway generates a token for each outgoing flow satisfying the corresponding guard-condition. Thus, since the guard-conditions evaluate to true when there is a violation, the OR-gateway will generate, for each detected violation, a new token in the corresponding outgoing flows. Then, each of these tokens will trigger the execution of the activity that repairs the violation. After the activity's execution, another OR-gateway checks for more violations and generates the corresponding tokens. If no violation occurs, the OR-gateway generates a token in its default path, which leads to the end event, since no more repairs are needed.

For instance, when running the BPMN example of Figure 10, we start with only one token placed in the activity *fireAssistant*. This activity represents the event in the original process model that can lead to the violation of several constraints, and thus, to the execution of their repairing activities.

Once this initial activity is executed, the token reaches an OR-gateway. This OR-gateway checks if the assistant who has been fired was also a current professor; if this is the case, the activity *del_ Current+del_ Supervises* is executed. Once this is executed, another OR-gateway checks if, after removing the supervising relationships, some assistant has ended without a supervisor. If this is the case, the OR-gateway creates a new token for each unsupervised assistant, and thus, for each one of this tokens, the user has to choose between removing the unsupervised assistant, or adding a new supervisor to him/her. Note that the tokens that need to be spawned by an OR-gateway can be automatically generated by means of a query into the information base that obtains the data that violates a particular constraint.

The execution of the process terminates when all the tokens have reached the end events, or when one of them arrives into an error end event. In the first case, the process terminates because it has repaired all the violations and thus, the information base is valid again. In the second case, the process terminates because it has found a violation that cannot be repaired[4].

For our purposes, we do not commit the changes established by the execution of those activities until all the tokens have successfully reached the end-event. That is, all the updates are delayed to be applied in a unique transaction at the end of the execution of the repairing-process rather than one at a time. There are two reasons behind this: 1) to avoid information base rollbacks in case one of the tokens reaches an error event, 2) it is known that applying the events one at a time loses the information of the previously-applied events, which might result in changes which contradict past events (e.g., deleting, at the end of the process, a tuple that was in-

[4]Following the BPMN standard, we use the common behavior of terminating the whole process instance when we reach an unhandled error event. Other possibilities are allowed.

serted previously to repair some violation) [28]. In order to be able to check the constraints through queries, these delayed changes are temporally stored in some auxiliary tables, similarly to the views in [29].

5.2. Ensuring Execution Termination

The generated BPMN extension might have cycles. Without any doubt, this is a source of non-termination of the processes. That is, there is the theoretic possibility in which a user gets stuck in executing the BPMN in an infinite loop.

This non-termination is inherent to the problem we are tackling. Indeed, it is easy to see that undecidable problems, such as first-order satisfiability, can be reduced to our BPMN execution. Roughly speaking, any first-order set of constraints is (finitely) satisfiable iff there is a finite execution trace of the BPMN they generate. Since first-order finite satisfiability is not decidable, knowing whether a BPMN has a finite execute trace is also undecidable, which implies that some BPMN will have infinite execution traces.

Fortunately, the very same solutions that apply in first-order logics also apply to our approach. In particular, there are multiple studies on finding decidable subsets of first-order logics which can be applied to our work. Intuitively, if we limit the constraints to be written in some particular subsets that guarantees decidibility, the generated BPMN will ensure the finiteness of its traces.

From the whole set of strategies for ensuring first-order logic decidability, we are interested on those based on showing the possibility of always building a finite database state that satisfies them. Other strategies exist, such as those used in Description Logics, but they do not match our particular purposes. Take, for instance, DL-Lite [30]. DL-Lite constraints' satisfiability is decidable, but its decidability does not come from ensuring the existence of finite instances, but comes from first-order logic rewritability, which means that it is possible to ensure the satisfiability of DL-Lite schemas by executing an SQL query. In fact, DL-Lite schemas can be infinitely satisfiable without being finitely satisfiable [31].

The strategies we are interested in are, basically, finite-model property, and those based on the chase-algorithm termination. The subsets of first-order logics that enjoy *finite-model property* (FMP) are those subsets that ensure that, given a set of first-order constraints written in it, if these constraints are satisfiable, then, they are finitely satisfiable. In terms of our approach, writing some constraints in some subset that enjoys FMP means that the generated BPMN has, at least, one finite executable trace (that computes the finite instance that satisfies the constraints). An example of first-order logic subset enjoying FMP, and directly characterized in UML/OCL notation, is OCL-Lite [32].

FMP ensures that at least one finite executable trace exists, but it does not ensure that all possible executable traces are finite. To ensure that all possible executable traces are finite, we have to look for a more tight condition. In particular, chase-termination.

Roughly speaking, chase is an algorithm for building an instance that shows that a set of first-order constraints is (finitely) satisfiable. The basic idea is that, if we guarantee that the chase algorithm can build a finite instance that satisfies the given first-order constraints, then, the generated BPMN will also guarantee that its execution traces are also finite. The rationale behind this behavior is that the BPMN generated is just a constructive algorithm for building the information base state that satisfies the constraints, thus, imitating the chase behavior. In fact, every BPMN execution trace corresponds to a chase trace over the repair-generating dependencies. Thus, ensuring that all chase traces are finite also ensures that all BPMN traces will also be finite. A direct subset of UML/OCL that enjoys such condition is OCLuniv [23].

5.3. Prototype tool implementation

In order to show the feasibility of our approach, we have implemented a prototype tool by means of adapting our previous version of the OpExec Java library [11]. OpExec is a Java library capable of parsing and executing BPMN activities. Since OpExec is not meant to control the BPM flow neither provide a GUI (indeed, controlling the BPM flow and bringing a GUI is a different problem [33]), we have to simulate the BPM flow of the original process programmatically. For the BPM extensions, however, we have extended OpExec to parse and execute the condition gateways that check the current information base state, and lead the execution to the corresponding next activity. This adaptation can be downloaded at `http://www.essi.upc.edu/~xoriol/opexec/`.

Using this library, a BPM-user can effectively repair the violations that take place when executing its activities, such as those discussed as examples in our paper. In particular, the library detects the violations and automatically applies, consecutively, the necessary activities to reach a new consistent information base state. Another example, ready to be executed, can be found at the given web page.

The purpose of this library is only meant to show that our solution can be implemented in practice, and let the possibility to the interested reader to download and try a prototype implementation. In addition, due to the difficulty of finding BPMN diagrams with first-order constraints, our tool already provides one example to play with it. Since it is not possible to determine the efficiency of our method by bringing one unique example, we refer the interested reader to check the previous discussion from Section 5.2, where we discuss the complexity and the termination of the method using already known results from well-studied languages such as Description Logics and OCL.

6. Related Work

Given a process model, and the definition of its tasks or activities, this paper presents an approach to automatically generate the necessary structural events to ensure data consistency, and representing them by means of a BPMN diagram. Due to this, it is possible to make changes to the underlying data model, while keeping the business process model the same.

This section analyses other works in the areas of constraint repair, process compliance (considering data) and consistency between UML diagrams.

6.1. Constraint Repair

Constraint repair is an area close to our proposal, as it deals with the detection of constraint violations and how to repair them. The techniques described in [34, 35, 36] are able to incrementally evaluate constraints, and they could be apply to detect the cases in which an activity would lead to a constraint violation. However, they cannot derive the repairs that would need to be applied; hence, they would not work with an extended interpretation of operation contracts.

Closer to our proposal, the approach in [37] is able to automatically create operations to modify the instances of a schema, whereas the work of [38] can complete the behaviour of an operation with additional updates to satisfy the constraints. However, our approach can naturally encode structural events applied recursively (i.e., by means of a loop in the BPMN), whereas these approaches might hang because of infinitely unfolding the recursion into a single method.

6.2. Compliance in business process models

There are many approaches that deal with the correctness or compliance of artifact-centric business process models [39, 7, 40, 41, 2, 42]. All these approaches analyse the semantic correctness of the model, considering all its dimensions; however, they work with a strict interpretation of the activities, as they cannot generate the required updates to fulfill the integrity constraints of the data model.

The work of [29] proposes DB-nets as an intermediate layer between a data model and a process model. The goal of the DB-net is to ensure that updates to the database take place at a point in time where no constraints are violated. This approach is different to our work, since we automatically generate the required structural events to fulfill the constraints. Similarly, [43] connects a BPMN process with a data model with the goal of detecting potential data design flaws. Again, the focus is not on correcting data issues, but on detecting them.

Following a completely different approach, [44] automatically generates a compliant artifact-centric process model given certain rules. In this case, tasks can be executed at any time, and do not follow a particular order. It is also worth mentioning [45]. It deals with the recovery of interweaved process instances when there are complex relationships between them.

There are other works dealing with process compliance at design-time [46, 47] and runtime [48], but without considering data. However, [46] focuses on detecting violations of task order execution and, like we do in our work, proposes repairs.

6.3. Consistency between UML diagrams

In spite of the fact that our approach uses a UML class diagram and a BPMN diagram, the latter could be replaced by a UML activity diagram. For this reason, it is worth mentioning some works which, although not explicitly artifact-centric, take into consideration the consistency between different UML models. In particular, [49] focuses on the consistency between UML activity diagrams and class diagrams. However, it considers that the object flow acts as a precondition and postcondition of the tasks and, unlike our approach, does not consider the changes they make to the data.

On the other hand, [50] performs a systematic mapping review on this topic. It lists the different consistency rules found in the analyzed works. Of the rules dealing with the class, state machine and activity diagrams, none of them would be applicable to our approach, either because we do not use the constructs (e.g. object nodes, swimlanes) or because we define the models from an analytical point of view [51], and hence, they are not relevant (e.g. visibility of attributes, assigning responsibility to classes).

7. Conclusions

This paper presents an approach to generate repairs for the activities in artifact-centric business process models. Given a data model, a process model and the specification of the activities in the process, our approach is able to generate extensions to the activities in order to ensure that the integrity constraints in the data model are fulfilled.

As we have shown, the main advantage of our approach is that the data model can evolve independently from the business process model and the specification of the activities; i.e. a change in the data model does not necessarily imply changes in the process or its activities. Although we use a UML class diagram, a BPMN diagram and OCL operation contracts, our work can be used with any other model which can be translated into first-order logic.

We extend our previous work [14], by entering into more detail of the logics behind our approach and discussing termination when generating the BPM extensions. We also simplify the BPM extensions by adding an activity merging step.

As further work, we would like to analyze the usage of BPMN reasoning tools to simplify our generated BPMN diagrams. Another area of interest is the development of heuristics or an aid to help choose the best repair when there are different repair options available. Last but not least, we could apply formal reasoning techniques to check if a repairing activity leads to a situation where the remaining activities in the BPM cannot be executed.

. **Acknowledgements:** This work is partially supported by the REMEDiAL project, funded by Ministerio de Economia, Industria y Competitividad (TIN2017-87610-R); by the Generalitat de Catalunya (under 2017-SGR-1749); and by the European Union Horizon 2020 Programme through the ERC Advanced Grant WhiteMech (No. 834228) and the ICT-48 2020 project TAILOR (No. 952215).

References

[1] R. Hull, Artifact-centric business process models: Brief survey of research results and challenges, in: OTM 2008, Vol. 5332 of LNCS, Springer, 2008, pp. 1152–1163.

[2] B. B. Hariri, D. Calvanese, G. De Giacomo, A. Deutsch, M. Montali, Verification of relational data-centric dynamic systems with external services, in: PODS 2013, ACM, 2013, pp. 163–174.

[3] D. Cohn, R. Hull, Business artifacts: A data-centric approach to modeling business operations and processes, IEEE-BDE 32 (3) (2009) 3–9.

[4] B. Bagheri Hariri, D. Calvanese, G. De Giacomo, A. Deutsch, M. Montali, Verification of relational data-centric dynamic systems with external services, in: Proc. of PODS, 2013, pp. 163–174.

[5] F. Belardinelli, A. Lomuscio, F. Patrizi, Verification of agent-based artifact systems, J. Artif. Intell. Res. 51 (2014) 333–376.

[6] D. Calvanese, M. Montali, M. Estañol, E. Teniente, Verifiable UML artifact-centric business process models, in: Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014, Shanghai, China, November 3-7, 2014, 2014, pp. 1289–1298.

[7] M. Estañol, M. Sancho, E. Teniente, Ensuring the semantic correctness of a BAUML artifact-centric BPM, Information & Software Technology 93 (2018) 147–162.

[8] M. Weske, Business Process Management: Concepts, Languages, Architectures, Springer, 2007.

[9] M. Dumas, M. L. Rosa, J. Mendling, H. A. Reijers, Fundamentals of Business Process Management, Springer, 2013.

[10] OMG, Object Constraint Language - version 2.4, available at: `http://www.omg.org/spec/OCL/2.4/PDF` (2014).

[11] G. De Giacomo, X. Oriol, M. Estañol, E. Teniente, Linking data and BPMN processes to achieve executable models, in: 29th International

Conference on Advanced Information Systems Engineering, CAiSE 2017, 2017, pp. 612–628.

[12] E. Franconi, A. Mosca, X. Oriol, G. Rull, E. Teniente, Ocl$_{\mathrm{FO}}$: first-order expressive OCL constraints for efficient integrity checking, Software and Systems Modeling 18 (4) (2019) 2655–2678.

[13] D. Costal, M. Sancho, E. Teniente, Understanding redundancy in UML models for object-oriented analysis, in: Advanced Information Systems Engineering, 14th International Conference, CAiSE 2002, Proceedings, 2002, pp. 659–674.

[14] X. Oriol, G. De Giacomo, M. Estañol, E. Teniente, Automatic business process model extension to repair constraint violations, in: S. Yangui, I. B. Rodriguez, K. Drira, Z. Tari (Eds.), ICSOC 2019, Vol. 11895 of LNCS, Springer, 2019, pp. 102–118.

[15] A. Queralt, E. Teniente, Specifying the semantics of operation contracts in conceptual modeling (2006) 33–56.

[16] P. Andrews, An Introduction to Mathematical Logic and Type Theory, Applied Logic Series, Springer, 2002.

[17] A. Queralt, E. Teniente, Verification and validation of conceptual schemas with ocl constraints, ACM Trans. Softw. Eng. Methodol. 21 (2) (2012) 13:1–13:41.

[18] X. Oriol, E. Teniente, Simplification of UML/OCL schemas for efficient reasoning, Journal of Systems and Software 128 (2017) 130–149.

[19] X. Oriol, E. Teniente, A. Tort, Computing repairs for constraint violations in UML/OCL conceptual schemas, Data Knowl. Eng. 99 (2015) 39–58.

[20] M. Estañol, E. Marcos, X. Oriol, F. J. Pérez, E. Teniente, J. M. Vara, Validation of service blueprint models by means of formal simulation techniques, in: Service-Oriented Computing, ICSOC 2017, Springer, 2017, pp. 80–95.

[21] X. Oriol, E. Teniente, Adapting integrity checking techniques for concurrent operation executions, in: P. Fonseca i Casas, M.-R. Sancho, E. Sherratt (Eds.), System Analysis and Modeling. Languages, Methods, and Tools for Industry 4.0, Springer International Publishing, Cham, 2019, pp. 235–248.

[22] J. W. Lloyd, R. W. Topor, Making prolog more expressive, The Journal of Logic Programming 1 (3) (1984) 225–240.

[23] X. Oriol, E. Teniente, $\mathrm{OCL}_{\mathrm{univ}}$: Expressive UML/OCL conceptual schemas for finite reasoning, in: 36th International Conference on Conceptual Modeling, ER 2017, 2017, pp. 354–369.

[24] A. Olivé, Integrity constraints checking in deductive databases., in: VLDB, Citeseer, 1991, pp. 513–523.

[25] A. Borgida, J. Mylopoulos, R. Reiter, On the frame problem in procedure specifications, IEEE Trans. Software Eng. 21 (10) (1995) 785–798.

[26] ISO, ISO/IEC 19510:2013 Information technology – Object Management Group Business Process Model and Notation (2013).

[27] R. M. Dijkman, M. Dumas, C. Ouyang, Semantics and analysis of business process models in BPMN, Information & Software Technology 50 (12) (2008) 1281–1294.

[28] E. Teniente, A. Olivé, Updating knowledge bases while maintaining their consistency, VLDB J. 4 (2) (1995) 193–241.

[29] M. Montali, A. Rivkin, Db-nets: On the marriage of colored petri nets and relational databases, T. Petri Nets and Other Models of Concurrency 12 (2017) 91–118.

[30] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, R. Rosati, Dl-lite: Tractable description logics for ontologies, in: AAAI, Vol. 5, 2005, pp. 602–607.

[31] R. Rosati, Finite model reasoning in dl-lite, in: European Semantic Web Conference, Springer, 2008, pp. 215–229.

[32] A. Queralt, A. Artale, D. Calvanese, E. Teniente, Ocl-lite: Finite reasoning on uml/ocl conceptual schemas, Data & Knowledge Engineering 73 (2012) 1–22.

[33] E. Diaz, J. I. Panach, S. Rueda, O. Pastor, Towards a method to generate gui prototypes from bpmn, in: 2018 12th International Conference on Research Challenges in Information Science (RCIS), 2018, pp. 1–12.

[34] A. Uhl, T. Goldschmidt, M. Holzleitner, Using an OCL impact analysis algorithm for view-based textual modelling, ECEASST 44 (2011).

[35] G. Bergmann, Translating OCL to graph patterns, in: MODELS 2014, Vol. 8767 of LNCS, Springer, 2014, pp. 670–686.

[36] J. Falleri, X. Blanc, R. Bendraou, M. A. A. da Silva, C. Teyton, Incremental inconsistency detection with low memory overhead, Softw., Pract. Exper. 44 (5) (2014) 621–641.

[37] M. Albert, J. Cabot, C. Gómez, V. Pelechano, Automatic generation of basic behavior schemas from uml class diagrams, Software & Systems Modeling 9 (1) (2010) 47–67.

[38] J. A. Pastor, A. Olivé, Supporting transaction design in conceptual modelling of information systems, in: J. Iivari, K. Lyytinen, M. Rossi (Eds.), Advanced Information Systems Engineering, 7th International Conference, CAiSE'95, Jyväskylä, Finland, June 12-16, 1995, Proceedings, Vol. 932 of LNCS, Springer, 1995, pp. 40–53.

[39] D. Calvanese, S. Ghilardi, A. Gianola, M. Montali, A. Rivkin, From model completeness to verification of data aware processes, in: C. Lutz, U. Sattler, C. Tinelli, A. Turhan, F. Wolter (Eds.), Description Logic, Theory Combination, and All That - Essays Dedicated to Franz Baader on the Occasion of His 60th Birthday, Vol. 11560 of Lecture Notes in Computer Science, Springer, 2019, pp. 212–239.

[40] D. Borrego, R. M. Gasca, M. T. G. López, Automating correctness verification of artifact-centric business process models, Information & Software Technology 62 (2015) 187–197.

[41] P. Gonzalez, A. Griesmayer, A. Lomuscio, Verification of GSM-based artifact-centric systems by predicate abstraction, in: A. Barros, D. Grigori, N. C. Narendra, H. K. Dam (Eds.), Service-Oriented Computing - 13th International Conference, ICSOC, Vol. 9435 of LNCS, Springer, 2015, pp. 253–268.

[42] I. Weber, J. Hoffmann, J. Mendling, Beyond soundness: on the verification of semantic business process models, Distributed and Parallel Databases 27 (3) (2010) 271–343.

[43] C. Combi, B. Oliboni, M. Weske, F. Zerbato, Conceptual modeling of inter-dependencies between processes and data, in: H. M. Haddad, R. L. Wainwright, R. Chbeir (Eds.), Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018, ACM, 2018, pp. 110–119.

[44] N. Lohmann, Compliance by design for artifact-centric business processes, in: S. Rinderle-Ma, F. Toumani, K. Wolf (Eds.), Business Process Management, Springer, Berlin, Heidelberg, 2011, pp. 99–115.

[45] H. Qin, G. Kang, L. Guo, Maxinstx: A best-effort failure recovery approach for artifact-centric business processes, in: S. Basu, C. Pautasso, L. Zhang, X. Fu (Eds.), Service-Oriented Computing, Springer, Berlin, Heidelberg, 2013, pp. 558–566.

[46] A. Awad, S. Smirnov, M. Weske, Resolution of compliance violation in business process models: A planning-based approach, in: OTM 2009, Vol. 5870 of LNCS, Springer, 2009, pp. 6–23.

[47] A. Elgammal, O. Türetken, W. van den Heuvel, M. P. Papazoglou, Root-cause analysis of design-time compliance violations on the basis of property patterns, in: ICSOC 2010, Vol. 6470 of LNCS, 2010, pp. 17–31.

[48] F. M. Maggi, M. Montali, M. Westergaard, W. M. P. van der Aalst, Monitoring business constraints with linear temporal logic: An approach based on colored automata, in: BPM 2011, Vol. 6896 of LNCS, Springer, 2011, pp. 132–147.

[49] R. Eshuis, Symbolic model checking of UML activity diagrams, ACM Trans. Softw. Eng. Methodol. 15 (1) (2006) 1–38.

[50] D. Torre, Y. Labiche, M. Genero, M. Elaasar, A systematic identification of consistency rules for UML diagrams, J. Syst. Softw. 144 (2018) 121–142.

[51] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition), Prentice Hall PTR, USA, 2004.