Automatic Business Process Model Extension to Repair Constraint Violations

Xavier Oriol[2], Giuseppe De Giacomo[1], Montserrat Estañol[2,3], Ernest Teniente[2]

[1] Sapienza Università di Roma, Rome, Italy
degiacomo@dis.uniroma1.it
[2] Universitat Politècnica de Catalunya, Barcelona, Spain
{oriol,estanyol,teniente}@essi.upc.edu
[3] Barcelona Supercomputing Center, Barcelona, Spain

Abstract. Consider an artifact-centric business process model, containing both a data model and a process model. When executing the process, it may happen that some of the data constraints from the data model are violated. Bearing this in mind, we propose an approach to automatically generate an extension to the original business process model that, when executed after a constraint violation, repairs the contents of the data leaving it in a new consistent state.

Keywords: BPMN, UML, Data-Aware Processes, Integrity constraints repair

1 Introduction

Artifact-centric business process modeling has been recognized as an appropriate approach to specify the two main assets of any organization, i.e. information (data as defined through the artifacts managed by the business) and processes (services offered by the organization to perform its business) [14, 15].

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 at least a conceptual model for data, such as a UML class diagram [11], and a model for the processes, such as BPMN [8, 28]. Linking data and processes along these two models has shown to be a feasible and practical way to achieve automatic executability of BPMs [5].

Furthermore, a data model always includes a set of *integrity constraints*, i.e. conditions that each state of the information base must satisfy. These constraints can be specified either graphically (such as multiplicity constraints) or textually (for instance by means of OCL constraints or SQL assertions).

The BPM states the order of execution of activities to successfully perform a business and also the effect of each executed activity over the contents of the information base (i.e. the object insertions and deletions performed by that activity over the classes in the data model). Clearly, this effect might violate some of the constraints in the data model.

Handling integrity constraints in the data model itself provides several advantages over manually programming them inside the BPM. Indeed, each constraint can be violated by different activities, and it is very difficult to manually identify all possible situations that may induce such violations. This makes programming manually the treatment of constraints an error-prone task and, thus, it should be avoided as much as possible. Therefore, we assume here that the execution of an activity in the BPM can raise an integrity constraint violation. A naive approach to deal with these violations would consist in forbidding the execution of the activity that caused the violation. However, this is not always appropriate because the actions entailed by the activity might have already happened in the real world. Thus, not performing this activity would end up with an information system that no longer represents the real world.

To overcome this situation, there is an alternative approach aimed at repairing the constraint violation, so that the activity can be executed anyway. This is achieved by means of performing additional updates, other than the ones explicitly specified by the activity. Therefore, under this approach, both the state of the real world and the contents of the information system will coincide and be consistent.

Since constraints can be repaired in several ways, the user (i.e. the person executing the process) should choose the most appropriate action in each situation. However, the chosen repair might lead to another violation which, in turn, requires additional repairing. Choosing repairs blindly can make the user get into a complicated sequence of violations/repairs which, known in advance, would have led him/her to make a better decision in the first place.

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 that shows all possible ways to repair it. Then, the user 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 user 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 the previously mentioned data model, we can automatically compute, at compile time, the whole chain of activities that, when executed, repairs a constraint violation. Therefore, we can extend the original BPM model by considering the flow of additional activities that have to be performed to preserve an integrity constraint. This extension can be computed for each activity of the original BPM and, since the computation can be done at compile time, it does not negatively impact the performance of the original process execution.

Moreover, by modelling the repairing process as a BPM, the process designer may customize these models at compile time to forbid some undesired paths/activities. Then, the final process model obtained can be used at execution time, by the user, to repair constraint violations when they occur. In particular, the user executes the original process as usual (e.g., through a CASE tool). However, when an integrity violation is detected, the current execution of the process stops, and the user starts executing the corresponding BPM extension to repair the violation caused by the last activity execution. When, the user has finished executing the extension, he/she can continue executing the original BPM, with the guarantee that no constraint is being violated.

2 Generating violation handling extensions in BPM

We will illustrate our approach by means of the BPMN diagram in Figure 1, together with the UML class diagram in Figure 2. The UML diagram specifies that employees work in and manage projects. Additionally, there is a *subset* constraint stating that each manager of a project should work in it.

The process model begins when creating a new project. Then, the user can either provide employee information, in order to add him to the project (*Add Employee*) or not (the process ends). If the former, the user then can choose to provide (another) project in order to delete this employee from it (*Delete From Project*), or not. In any case, the flow goes back to the option of adding another employee to the recently created project. Hence, the message events in the BPMN diagram correspond to user-provided input, and not to evaluation of expressions using process data. This is why we use Event-Based Gateways. Note that there may be other processes in the company for hiring managers or performing other tasks, which are not shown here.

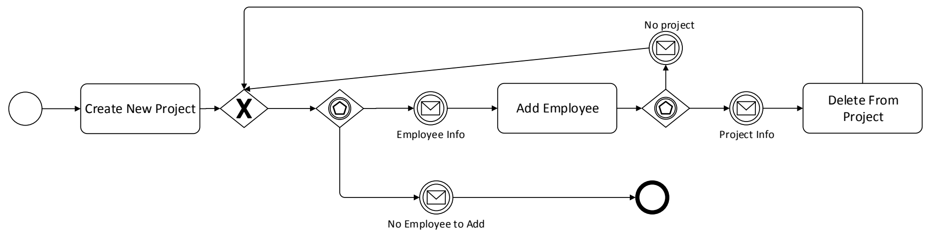

Fig. 1. Initial BPMN diagram

When executing any process activity, a violation of a constraint can occur. For example, when executing *Delete From Project* the *subset* constraint may be violated (i.e. a manager of the project is not one of its employees any more). Naturally, we could reject such activity to avoid the violation, but it may likely happen that this deletion has already taken place in the real world and cannot be undone. Thus, a reactive behavior has to be applied. We can repair that constraint by removing this employee as a manager of the project. However, this additional removal might in turn violate the minimum cardinality constraint stating that each project needs at least one manager, thus forcing the execution of more updates to preserve the consistency of the information base. This is the difficulty of the problem of integrity constraint repairing.

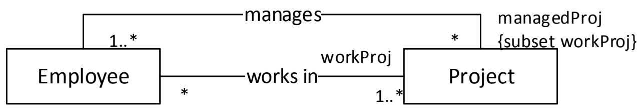

Fig. 2. Class diagram for the BPMN diagram in Figure 1

Fortunately, the constraints that might be violated when repairing other constraints can be determined at compile time; i.e., we can identify them by inspecting the constraints' definition itself, without considering the information contents. Indeed, several approaches build a dependency-graph showing this relation among the constraints [21,24]. Thus, the idea is that, to repair a constraint violation C and 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).

For instance, in our example we would be able to determine that the subset constraint might violate the minimum cardinality constraint requiring one manager for each project. Thus, after repairing the subset constraint, we should check, and possibly repair, such cardinality constraint.

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 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 being 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 a database state and a data update, derive new updates that must be applied to repair a constraint violation [22]. 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 a BPMN activity, its execution might 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 BPMN activities and the dependency-graph edges determine the flow between them.
5. *Customization.* Finally, the BPM designer may decide to prune some of the suggested ways to repair a constraint in the BPM. Indeed, our method generates all possible activities that might be applied to repair a violation. However, it might be the case that some of them are not desirable in the domain. In this step, we show how to prune undesired repairs.

In this way, given any BPM activity, we compute its BPM extension which guarantees that, when executed, it checks and repairs all violations that might occur. This extension could be integrated in the original BPM through a CASE tool, and can be used at runtime to repair constraint violations through a process executor, such as [5]. In this paper, we leave the part of showing the extension as further work, and concentrate on generating the extension and executing it. Furthermore, although we use the BPMN and UML notations, it is worth to mention that other languages, such as *service blueprints* for instance, might be used as long as they are detailed enough to be executed [10]. In particular, we only need these notations to be translatable into first-order logics, which is the base framework of our approach. Finally, one limitation of our approach is that two tasks cannot be executed simultaneously, as they might interact to cause a constraint violation. In such cases, they should be serialized.

2.1 Translating constraints into RGDs

RGDs are logic formulas that, given a database state and a set of updates, derive new updates in order to repair a constraint violation [22]. Every UML/OCL constraint gives raise to several RGDs, each one capturing a different way to violate/repair it. For instance, consider the subset constraint stating that if x *Manages* y, then x *WorksIn* y. This constraint gives raise to the following RGDs:

$$\iota Manages(x,y) \wedge \neg WorksIn(x,y) \rightarrow \iota WorksIn(x,y) \quad (1)$$

$$Manages(x,y) \wedge \delta WorksIn(x,y) \rightarrow \delta Manages(x,y) \quad (2)$$

$$\iota Manages(x,y) \wedge \delta WorksIn(x,y) \rightarrow \bot \quad (3)$$

The first RGD states that if we insert that x *manages* y ($\iota Manages(x,y)$), when x *is not working in* y ($\neg WorksIn(x,y)$), then, we must insert that x *works in* y ($\iota WorksIn(w,y)$) to guarantee the consistency of the new state after the update. Similarly, the second RGD states that if we delete some worker x from y, s.t. x was managing y, then, we must also delete that x no longer manages y. Finally, the third RGD asserts that, if we insert some manager x into y and we delete x as working in y, there is an irreparable violation, since these are two contradictory events (a manager of a project should work in it) that are executed simultaneously. Generally, any RGD with $\bot$ in the head cannot be repaired.

Not all RGDs are deterministic since some violations can be repaired in different ways. RGDs capture this indeterminism through disjunctions and existential variables in their head. E.g., consider the cardinalities from our UML diagram stating that each employee works at least in one project, and that each project has, at least, one manager. These constraints give raise, respectively, to the following RGDs:

$$Employee(x) \wedge \delta WorksIn(x,y) \wedge \neg OtherWorksIn(x) \rightarrow \delta Employee(x) \vee \iota WorksIn(x,y') \quad (4)$$
$$OtherWorksIn(x) \leftarrow WorksIn(x,z) \wedge \neg\delta WorksIn(x,z)$$
$$Project(y) \wedge \delta Manages(x,y) \wedge \neg OtherManager(y) \rightarrow \delta Project(y) \vee \iota Manages(x',y) \quad (5)$$
$$OtherManager(y) \leftarrow Manages(z,y) \wedge \neg\delta Manages(z,y)$$

Intuitively, RGD 4 detects a violation when we delete employee x from project y, and x does not work for any other project. In this case, we should choose between deleting the employee x, or adding a new project y' where he is working. Note that the decision is indeterministic. Moreover, choosing the project y' is also indeterministic since it can take different values. A similar condition with projects and managers is stated by RGD 5.

Some RGDs can be simplified by taking into account that, given a particular domain, some events cannot ever happen. Indeed, we can consider that projects are never deleted from the system. Thus, the literal $\delta Project$ can be safely deleted from the head of the RGD 5, leading to a new formula:

$$Project(y) \wedge \delta Manages(x,y) \wedge \neg OtherManager(y) \rightarrow \iota Manages(x',y) \quad (6)$$
$$OtherManager(y) \leftarrow Manages(z,y) \wedge \neg\delta Manages(z,y)$$

The structural updates (i.e., insertions or deletions) that cannot happen in a domain can be extracted from the UML class diagram itself. When a class/association A is considered to be *add-only*, this means that the event of deleting an instance of A cannot take place. Similarly, if a class/association A is considered to be *frozen*, no insertion nor deletion update can occur in its population.

The problem of obtaining RGDs from UML/OCL constraints, and simplifying them, is already solved in [22]. For our purposes, we consider only constraints written in the UML/OCLuniv subset [20]. Roughly speaking, UML/OCLuniv is the subset of UML/OCL where all constraints are universally quantified (i.e., no OCL *exists* operator is allowed), with the exception of UML min. cardinalities. We impose this limitation because: 1) RGDs from UML/OCLuniv constraints generate repairs of only one single structural event, which are easier to translate to BPMN activities, and 2) the termination of the repair process is guaranteed (while, in general OCL constraints, the repair process is undecidable) [20].

2.2 Building the dependency-graph of RGDs

Given a set of RGDs, we can build a dependency-graph that shows which RGDs may trigger other RGDs. Indeed, consider the case in which we delete some worker x from project y, when x was manager of y. In this situation, RGD 2 states that we have to additionally delete x as a manager of y. However, if we do so, it might be the case that the project y has no manager, thus triggering RGD 6, which states that we should add a new manager to it.

This *triggering relationship* between RGDs can be depicted graphically in several ways. For our purposes, we choose the one from [21]. Briefly, for each

RGD, the left-hand side is depicted as a vertex (called *constraint-vertex*), and the different structural events from the right-hand side are depicted also as vertices (called *repair-vertex*). There is an arrow from an RGD constraint-vertex to each of its corresponding repair-vertices that indicates that, when the condition stated in the constraint-vertex is satisfied, one of its repair-vertices should be executed. Then, there is also an edge from a repair-vertex to each of the constraint vertices that might have been violated because of its execution.

The grey part of Figure 3 shows the dependency-graph of RGDs 1, 2, 3, 4 and 6, together with other RGDs that will be used in the rest of the paper[4]. Note that there is, as expected, a triggering relationship between 2 and 6.

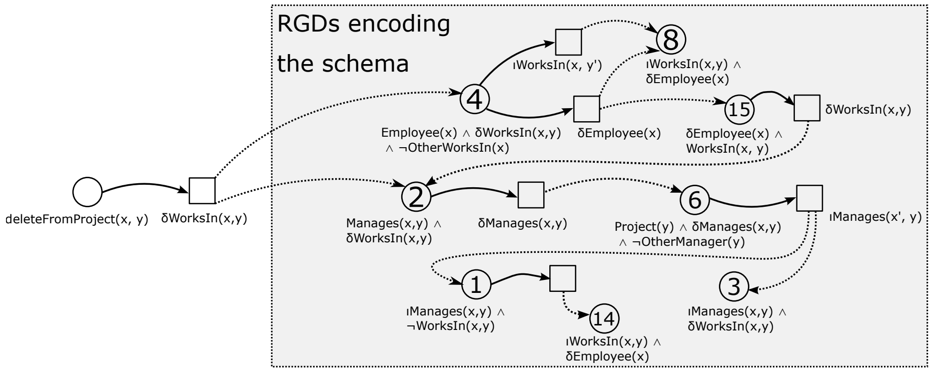
RGDs encoding the schema
deleteFromProject(x, y)
δWorksIn(x,y)
4
Employee(x) ∧ δWorksIn(x,y) ∧ ¬OtherWorksIn(x)
ιWorksIn(x, y')
δEmployee(x)
8
ιWorksIn(x,y) ∧ δEmployee(x)
15
δEmployee(x) ∧ WorksIn(x, y)
δWorksIn(x,y)
2
Manages(x,y) ∧ δWorksIn(x,y)
δManages(x,y)
6
Project(y) ∧ δManages(x,y) ∧ ¬OtherManager(y)
ιManages(x', y)
1
ιManages(x,y) ∧ ¬WorksIn(x,y)
14
ιWorksIn(x,y) ∧ δEmployee(x)
3
ιManages(x,y) ∧ δWorksIn(x,y)

Fig. 3. Fragment of the dependency-graph showing constraint-vertices as circles, repair-vertices as squares, and triggering relationships with dashed-edges between both.

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 a structural event (i.e. an update operator) in common. Indeed, this means that the repair of the first constraint is applying some update that can potentially violate the second constraint. We could also apply some optimizations to remove some triggering relationships [21], but we leave them out due to space limitations.

2.3 Associating activities to the dependency-graph

We start with a graph showing which constraints can be violated when repairing other constraints, and we want to know now which constraints can be violated when executing a BPMN activity in the process model. This is achieved by specifing the BPMN activity as an RGD, include this RGD in the dependency-graph, and identify the RGDs in the original graph reachable from it.

A BPMN activity can be seen as an RGD whose repair is, in fact, the execution of the update it specifies. For instance, consider the BPMN activity *delete from project*, from Figure 1, stating the deletion of an employee from a project given by parameter. This BPMN activity can be written as the RGD:

[4] We do not include all the generated RGDs for easier understandability.

$$\mathit{deleteFromProject}(x,y) \rightarrow \delta\,\mathit{WorksIn}(x,y) \tag{7}$$

This way of specifying BPMN activities is already used in [5], where an automatic translation from BPMN activities written with OCL constraints into these RGDs is given. Now, this RGD can be incorporated in the dependency-graph, as shown in Figure 3, and indicate its triggering relationships, i.e. those RGDs with a constraint-vertex containing the structural events applied in the BPMN activity. In our running example, this new RGD would point to RGDs 2 and 4 since they have the $\delta\,\mathit{WorksIn}$ predicate in common.

Thus, the RGDs possibly affected by the execution of the BPMN activity correspond to the fragment of the dependency-graph reachable from the RGDs encoding the activity. In our example, these correspond to the RGDs seen so far, but, in a real case, they would likely be a subset of all the RGDs in the graph. Intuitively, this is because the execution of some activity affecting one part of the diagram will not necessarily propagate its effects to the whole diagram.

The rest of the RGDs, i.e., those that are not reachable from the RGD encoding the activity, are removed. They correspond to constraints that can never be violated when executing and repairing the main activity. Thus, they can be safely removed from the graph.

2.4 Translating the dependency-graph into a BPMN diagram

Now, we translate the relevant part of the pruned dependency-graph we have just obtained into a BPMN diagram. 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, and any repair-vertex becomes a single BPMN activity that applies the repair itself. Then, these BPMN 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 repair applied, or by an end-event in case none of the constraints can be actually 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 which means that, if we reach the violation of such constraint in the way captured by the constraint-vertex, 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 its unique repair. If there is more than one potential repair, the constraint-vertex is translated to an event-gateway that enables the user to choose 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 user input to choose the value for the existential variables. In this case, the BPMN activity is represented as a *receive task*. As an example, consider the case of RGD 2 where we have a repair vertex which inserts a new Manager x' to the project. In case a user wants to repair this RGD by means of this x' insertion, we need the user to explicitly choose a specific value for this x'.

```
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)
}}}
```

Fig. 4. Algorithm for obtaining the BPMN diagram from the dependency-graph (Java-like notation used)

After applying a repair, it may be the case that other constraints are violated. If this is the case, *several* constraints may need to be repaired. The OR-gateway is in charge of checking this. If no violation occurs, the flow continues to the end event. Otherwise, one (or several) path(s) will be activated. These paths will lead to the corresponding contraint-vertices so that the violations can be repaired.

This is guaranteed by the guard conditions in the OR-gateway's outgoing flows. That is, an outgoing flow pointing to the (BPMN translation of a) constraint vertex c has, as a guard, the logic condition encoded in c. Thus, the unique way to execute an activity that repairs a violation (or leads to an error event) is through the guard that first checks the constraint. So, these activities only take place when the update needs to be applied.

Note that we do not use OR-joins for synchronizing the activities execution. Intuitively, such synchronization is not necessary since each path execution represents a different violation repair for some particular values, and such repair for those particular values is independent from the rest of violations/repairs. We capture this behaviour using OR-gateways without OR-joins for ease of readability. However, if the user prefers to avoid this kind of diagrams, since OR-gateways are usually synchronized with OR-joins, our method can be adapted to replace these OR gateways by a combination of XORs and tasks.

The translation process of our approach is formalized in Algorithm 4. This algorithm has two input parameters: the (relevant part of) the dependency graph, together with the constraint-vertex representing the BPMN activity that triggers all the repairing procedure, and thus, behaves as the starting activity. As output, the algorithm provides the resulting BPMN diagram. It is easy to see that the algorithm runs in polynomial time w.r.t the input.

In Figure 5 we show the result of applying the previous algorithm to our running example. In this BPMN we see that when executing *deleteFromProject* it may happen that we satisfy all the constraints, or that we need to delete the worker as a manager (to satisfy the subset constraint), or that we need to choose between: 1) deleting him as an employee or 2) including him in a new project (to satisfy the minimum cardinality constraint stating that each employee works in at least one project).

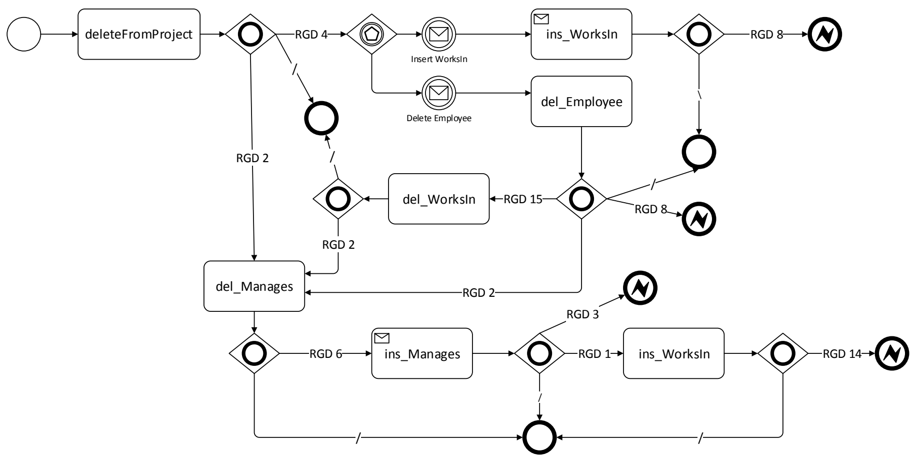

Fig. 5. BPMN diagram for repairing activity deleteFromProject in case of a violation

Although it does not happen in our running example, we should note that the BPMN diagram might have cycles. This is the case when a constraint C_1 can be repaired in such a way that violates a constraint C_2, and when repairing C_2 we might end violating C_1 again. These cycles require special attention since, in the general case, they are a source of an infinite BPM execution.

However, limiting the constraint language to be UML/OCLuniv ensures that, at runtime, these cycles do not execute forever. That is, at some point, the guard that checks if one of these activities has to be executed is going to be false, and thus, the user will not be able to loop forever. Roughly speaking, this is because, when repairing a UML/OCLuniv constraint violated by some object a of class A, we only need to create instances of a class different than A. Moreover, if those other instances violate another constraint, it is guaranteed that they will create instances of a class different than A and their current one. In general, when repairing UML/OCLuniv constraints we will never create instances of already visited classes. Thus, since the number of different classes is finite, all the constraints will eventually be repaired in a finite number of steps. A more detailed explanation of the finiteness of the computation, based on the chase algorithm termination, is given in [20].

2.5 Customization

The obtained BPMN diagram represents all possible ways to repair the various constraints that can eventually be violated by the activity execution. This is due to the fact that RGDs capture all possible ways to repair a constraint [22], 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 may consider inappropriate to fire employees just to repair a constraint. In this context, we want to avoid this kind of repair.

To do this, we need to consider which RGDs result in deleting employees. 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 delete employees and all the subsequent ones. In this case, removing *del_ Employee* causes the deletion of *del_ WorksIn* and *del_ Manages* since they cannot be reached from the starting BPMN node. This leads to the final BPMN diagram shown in Figure 6.

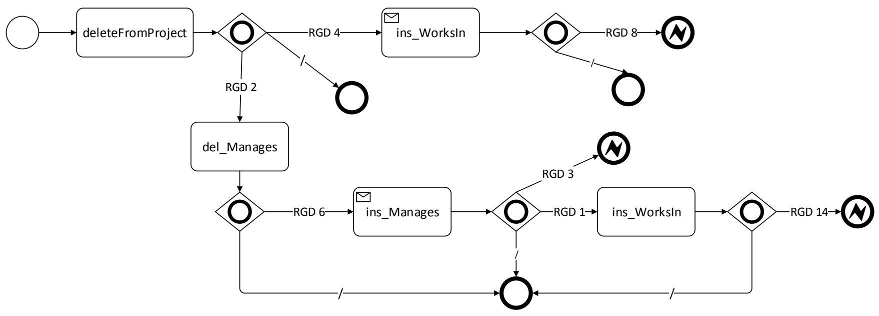

Fig. 6. Customized BPMN diagram for repairing the deleteFromProject activity

As a result of the whole process, we have obtained a BPM diagram (referred to as *BPM extension*) that, executed after the *Delete From Project* activity from Figure 1, ensures that *Delete From Project* preserves the consistency of the data regarding the constraints in Figure 2. Note that the execution of the original diagram is paused while the BPM extension executes to repair the constraints. Finally, this extension could be directly embedded in the original diagram, or be shown only on-demand, i.e., when some violation occurs, in order to guide the user to repair the violations due to this activity.

3 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 use an existing BPM executor to run our generated extension to show the feasibility of our approach.

3.1 Business process extension execution semantics

Intuitively, the BPMN language is based on token semantics [16]. 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 activates 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 [7].

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 6, we start with only one token placed in the activity *deleteFromProject*. This activity represents the structural 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 employee who has been unassigned from the project is assigned to another project; if this is not the case, the activity *ins_ WorksIn* is executed. The OR-gateway also checks if the employee was the manager of the initial project, and if this is the case, the *del_ Manages* activity is executed.

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 database is valid again. In the second case, the process terminates because it has found a violation that cannot be repaired[5].

It is worth mentioning that, in our approach, we consider that an OR-gateway can generate several tokens pointing to the same activity. This is the case when a constraint is violated several times by means of several data. For instance, consider that the *deleteFromProject* activity, instead of just deleting one worker given by parameter from some project, deletes several workers from different projects (i.e., its input parameter is a list of workers instead of just one). In this case, we might need to execute the activity *del_Manages* several times (one time for each worker that was also managing the project), similarly to [4]. In any case, note that the tokens that need to be spawned by an OR-gateway can be automatically generated by means of a query into the database that obtains the data that violates a particular constraint.

For our purposes, we do not commit the database 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 database 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 inserted previously to repair some violation) [25]. In order to be able to check the constraints through database queries, these delayed changes are temporally stored in some auxiliary database tables.

3.2 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 [5]. 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 [6]), 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 checks the current database state, and leads 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. For instance, consider the case of two different

[5] 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 [16].

employees working and managing two different projects. In such case, removing the first employee from his project leads to a constraint violation. Our adapted library detects this situation, and forces the execution of the activities which make the data consistent again. That is, it applies the sufficient activities to remove the first employee from the first project, adds it to the second project, removes him as a manager from the project, picks the second employee and makes her manager of the first project, and includes her as a worker from the first project. A test file to check this behavior is available at the previous website.

4 Related Work

There are several approaches to model artifact-centric BPM, ranging from more flexible approaches [17] - which use condition-action rules instead of a BPMN, for instance - to procedural ones, such as ours, which establish a clear order for task execution [4, 11, 27]. To the best of our knowledge, there are no previous works which deal with constraint repair in artifact-centric process models and which generate an extension of the original model to carry out the repairs.

4.1 Constraint Repair

In the conceptual modeling literature, there are quite a lot of proposals for incrementally evaluating constraints [3, 12, 26]. Using these techniques, it is possible to efficiently identify when the execution of some activity leads to a constraint violation. However, none of them is able to derive the repairing activities that need to be applied, as we do.

In a different way, some approaches are meant to, given a schema with some constraints, build operations for inserting/deleting/updating instances in the schema, and completing the behavior of the operation with additional updates to satisfy all the constraints [1, 23].

However, we argue that these approaches of compiling all the repairing actions into a single activity are more limited than ours. Indeed, our approach generates a process, rather than a single activity, and a process can naturally encode recursive repair actions by means of adding a cycle in the BPMN diagram. However, the proposals defined in [1, 23] lacks recursion, thus, these approaches might hang because of infinitely unfolding the recursion into a single method.

4.2 Compliance in business process models

There are several approaches to verify and/or validate the correctness of artifact-centric business process models, such as [4, 11, 13, 14, 27]. However, these works focus on the correctness of the model as a whole and checking if it fulfills certain desirable properties. Note that this is different from our proposal, where we detect potential integrity constraint violations and find ways to repair them.

Similarly, [18] applies constraint programming to detect errors in data constraints without the need for an information base, taking the data flow through the process into consideration. However, the approach does not generate repairs for these constraints.

On the other hand, there are other works dealing with process compliance at design-time [2, 9] and runtime [19], but without considering data. For instance, [2] focuses on detecting violations of task order execution and proposes repairs. Similarly, [9] checks the process's compliance with several patterns, such as existence, absence or separation-of-duties, determining the reason behind each violation. On the other hand, [19] detects constraint violations at runtime and proposes several strategies to deal with them, but does not generate any repairs.

5 Conclusions

We have proposed an approach to automatically extend a business process model to include, at compile time, the activities that might repair constraint violations. We take as a starting point an artifact-centric BPM, represented by a UML class diagram (with OCL integrity constraints) to model the data; and a BPMN diagram to model the tasks and their execution order.

As further work, we would like to study the generation of BPM extensions for full OCL constraints rather than OCLuniv ones, and to analyze the usage of BPMN reasoning tools to optimize 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.

References

1. Albert, M., Cabot, J., Gómez, C., Pelechano, V.: Automatic generation of basic behavior schemas from uml class diagrams. Software & Systems Modeling 9(1), 47–67 (Jan 2010)
2. Awad, A., Smirnov, S., Weske, M.: Resolution of compliance violation in business process models: A planning-based approach. In: OTM 2009. LNCS, vol. 5870, pp. 6–23. Springer (2009)
3. Bergmann, G.: Translating OCL to graph patterns. In: MODELS 2014. LNCS, vol. 8767, pp. 670–686. Springer (2014)
4. 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)
5. De Giacomo, G., Oriol, X., Estañol, M., Teniente, E.: Linking data and BPMN processes to achieve executable models. In: 29th International Conference on Advanced Information Systems Engineering, CAiSE 2017. pp. 612–628 (2017)
6. Diaz, E., Panach, J.I., Rueda, S., Pastor, O.: Towards a method to generate gui prototypes from bpmn. In: 2018 12th International Conference on Research Challenges in Information Science (RCIS). pp. 1–12 (May 2018)
7. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Information & Software Technology 50(12), 1281–1294 (2008)
8. Dumas, M., Rosa, M.L., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management. Springer (2013)
9. Elgammal, A., Türetken, O., van den Heuvel, W., Papazoglou, M.P.: Root-cause analysis of design-time compliance violations on the basis of property patterns. In: ICSOC 2010. LNCS, vol. 6470, pp. 17–31 (2010)

10. Estañol, M., Marcos, E., Oriol, X., Pérez, F.J., Teniente, E., Vara, J.M.: Validation of service blueprint models by means of formal simulation techniques. In: Service-Oriented Computing, ICSOC 2017. pp. 80–95. Springer (2017)
11. Estañol, M., Sancho, M., Teniente, E.: Ensuring the semantic correctness of a BAUML artifact-centric BPM. Information & Software Technology 93, 147–162 (2018)
12. Falleri, J., Blanc, X., Bendraou, R., da Silva, M.A.A., Teyton, C.: Incremental inconsistency detection with low memory overhead. Softw., Pract. Exper. 44(5), 621–641 (2014)
13. Gonzalez, P., Griesmayer, A., Lomuscio, A.: Verification of gsm-based artifact-centric systems by predicate abstraction. In: Service-Oriented Computing - 13th International Conference, ICSOC. pp. 253–268 (2015)
14. Hariri, B.B., Calvanese, D., De Giacomo, G., Deutsch, A., Montali, M.: Verification of relational data-centric dynamic systems with external services. In: PODS 2013. pp. 163–174. ACM (2013)
15. Hull, R.: Artifact-centric business process models: Brief survey of research results and challenges. In: OTM 2008. LNCS, vol. 5332, pp. 1152–1163. Springer (2008)
16. ISO: ISO/IEC 19510:2013 Information technology – Object Management Group Business Process Model and Notation (2013)
17. Leno, V., Dumas, M., Maggi, F.M.: Correlating activation and target conditions in data-aware declarative process discovery. In: Business Process Management. pp. 176–193. Springer International Publishing, Cham (2018)
18. López, M.T.G., Gasca, R.M., Pérez-Álvarez, J.M.: Compliance validation and diagnosis of business data constraints in business processes at runtime. Inf. Syst. 48, 26–43 (2015)
19. Maggi, F.M., Montali, M., Westergaard, M., van der Aalst, W.M.P.: Monitoring business constraints with linear temporal logic: An approach based on colored automata. In: BPM 2011. LNCS, vol. 6896, pp. 132–147. Springer (2011)
20. Oriol, X., Teniente, E.: OCL_{univ}: Expressive UML/OCL conceptual schemas for finite reasoning. In: 36th International Conference on Conceptual Modeling, ER 2017. pp. 354–369 (2017)
21. Oriol, X., Teniente, E.: Simplification of UML/OCL schemas for efficient reasoning. Journal of Systems and Software 128, 130–149 (2017)
22. Oriol, X., Teniente, E., Tort, A.: Computing repairs for constraint violations in UML/OCL conceptual schemas. Data Knowl. Eng. 99, 39–58 (2015)
23. Pastor, J.A., Olivé, A.: Supporting transaction design in conceptual modelling of information systems. In: 7th International Conference on Advanced Information Systems Engineering, CAiSE'95. pp. 40–53 (1995)
24. Queralt, A., Teniente, E.: Verification and validation of conceptual schemas with ocl constraints. ACM Trans. Softw. Eng. Methodol. 21(2), 13:1–13:41 (Mar 2012)
25. Teniente, E., Olivé, A.: Updating knowledge bases while maintaining their consistency. VLDB J. 4(2), 193–241 (1995)
26. Uhl, A., Goldschmidt, T., Holzleitner, M.: Using an OCL impact analysis algorithm for view-based textual modelling. ECEASST 44 (2011)
27. Weber, I., Hoffmann, J., Mendling, J.: Beyond soundness: on the verification of semantic business process models. Distributed and Parallel Databases 27(3), 271–343 (2010)
28. Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer (2007)