Linking Data and BPMN Processes to Achieve Executable Models

Giuseppe De Giacomo[1], Xavier Oriol[2], 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] SIRIS Lab, Research Division of SIRIS Academic, Barcelona, Spain

Abstract. We describe a formally well founded approach to link data and processes conceptually, based on adopting UML class diagrams to represent data, and BPMN to represent the process. The UML class diagram together with a set of additional process variables, called Artifact, form the information model of the process. All activities of the BPMN process refer to such an information model by means of OCL operation contracts. We show that the resulting semantics while abstract is fully executable. We also provide an implementation of the executor.

Keywords: BPMN, UML, Data-Aware Processes, Artifact-Centric Processes

1 Introduction

The two main assets of any organization are (*i*) information, i.e., data, which are the things that the organization knows about, and (*ii*) processes, which are collections of activities that describe how work is performed within an organization.

Obviously there is the need for representing and making explicit and precise the contents of these two assets. This has led to conceptual models for data, such as UML class diagrams [1], and conceptual models for processes, such as BPMN [2, 3]. Unfortunately these conceptual models are only rarely formally related [4, 5]. In fact, they are typically developed by different teams, the data management team and the process management team, respectively, which use their own models and methodologies. This leads to the development of two independent and unrelated designs and formalizations, one concerned with data and one with processes, while the interaction between the two is neglected [6, 7].

Moreover, when we arrive to tools for process simulation, monitoring and execution, the two aspects need to come together, and indeed all tools, such as BIZAGI STUDIO or SIGNAVIO, provide a typically proprietary way to realize the connection. However such a connection is essentially done *programmatically*, by defining an internal data model and associating it to the BPMN constructs in the process through suitable *business rules* expressed as actual code (e.g., written in JAVA) to detail what happens to the data and how data are exchanged with the users and other processes. Unfortunately, this way of connecting data and processes becomes elicited programmatically, but not conceptually.

Recent research is bringing forward the necessity of considering both data and processes as first-class citizens in process and service design [7–9]. In particular, the so called artifact-centric approaches, which advocate a sort of middle ground between a conceptual formalization of dynamic systems and their actual implementation, are promising to be quite effective in practice [6, 10, 11].

In this paper, inspired by artifact-centric approaches, we consider the case in which the data of the domain of interest of a given process are conceptually represented using a UML class diagram, while the process itself is described in BPMN. We adopt UML and BPMN as they are the standard and the most common formalisms for conceptual representation of data in software engineering and processes in BPM, respectively. In this way, we do not propose yet-another-formalism, but combine standard ones in a new integrated way to link data and processes. Other languages might be chosen as well as long as they have an unambiguous semantics, e.g. ORM/ER-diagrams for defining the data, or UML activity diagrams, as used for instance in [12], to define the process.

The key idea underlying our proposal is that, in order to link both formalisms, we propose also: (1) the notion of Artifact, which acts as a collection of process variables to be associated with a process instance, and (2) the specification of how the process activities refer and update the variables of the Artifact, or the domain data. Both concepts can be formally specified through standard languages that suitable accommodate our UML and BPMN diagrams. Indeed, the Artifact can be represented as a new class of the UML class diagram with its convenient attributes and associations to the rest of UML classes, and the process activities can be specified through OCL operation contracts. Again, 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 [13].

In this way, the executability of the overall framework can rely on relational SQL technology, since the data to insert/delete/return by each activity can be characterized through a relational-algebra query, and thus, an SQL statement. In particular, the UML class diagram is encoded as a relational database, the BPMN diagram as a Petri net, and the OCL contracts as logic rules that derive which SQL statements must be applied to the database when an activity is executed. As a proof of concept, we have developed a prototype, written in Java, which allows loading at compile time all the models in our framework and then execute their operations at run time in a relational database.

2 Preliminaries

UML class diagrams and their instances. A *UML class diagram* [1] is formed by a hierarchy of *classes*, n-ary *associations* among such classes (where some of them might be reified, i.e, *association classes*), and *attributes* inside these *classes*. In addition, a UML schema might be annotated with *minimum/maximum* multiplicity constraints over its association-ends/attributes, and hierarchy constraints (i.e., disjoint/complete constraints). In this paper, we use the notation $C \sqsubseteq C'$ to refer that C is a subclass of C'. We adopt a *conceptual perspective* (as opposed to a *software perspective*) of UML class diagrams, as typical of the analysis phase of the development process [14].

Moreover, for convenience, we assume that the UML class diagram contains only those features that can be mapped into SQL tables with primary/foreign key constraints. For example we express in the diagram optional/mandatory (min multiplicity 0 or 1), single/multivalued properties (max multiplicity 1 or *), but not, e.g., min/max multiplicity 3. All other expressions are assumed to be written and treated as OCL constraints (see below). A *UML class diagram instance* is a set of objects and relationships among such objects. Each object is classified as an *instance* of one or more UML classes, and each relationship as an *instance* of one UML association. We assume that, whenever an object o is classified as an instance of C, and $C \sqsubseteq C'$, then, o is also classified as an instance of C'. Note that this process of *completing* the classifications of an object can be automatically computed through a chase over the UML class hierarchy. This automatic mechanism is called *ISA closure*.

OCL. OCL [15] is a textual language for defining queries over a UML schema, whose result depends on the contents of its UML instance. In particular, OCL boolean expressions are widely used to define: (1) textual integrity constraints that should be satisfied by UML instances of the schema, (2) operation contracts pre/postconditions, that is, expressions that should be satisfied by the UML instances of some schema before/after executing some operation, and (3) queries specifying the return value of some operation. OCL expressions are usually tied to a particular context UML class. For instance, the OCL operation contract of a certain operation is tied to the class in which the operation is defined. In this situation, the OCL expression *self*, refers to the object in which the operation is invoked (in a similar way to the Java keyword *this*). Similarly, an OCL constraint tied to some class C uses *self* to refer to any instance of C.

The core idea underlying OCL is the notion of *navigation*. Given an OCL expression referring to an object, such as *self*, we can navigate to *objects/values* related to such object through some association/attribute using the name of the association-end/attribute we want to traverse. For instance, the OCL expression *self.album* tied to some context class *Artist* returns the albums related to the particular artist referred by *self*. A navigation can also be defined starting from an OCL expression referring to a collection of objects. For instance, the OCL expression *Artist.allInstances()* refers to the set of all *Artist* objects, thus, *Artist.allInstances().album* returns all the albums that can be obtained from all the artists. Moreover, due to this capability of navigating from collections, OCL permits chaining one navigation after another. For instance, *self.album.track* refers to all the tracks of all the albums of a particular *Artist self*. Given these navigations, OCL offers several OCL operators to obtain basic type values (such as boolean, or integer values), or other collections from them. For instance, *self.album.track->forAll(o|o.duration >0)* returns true iff all the durations of all the tracks o of some artist *self* are greater than 0.

We assume in this paper that all OCL expressions are written in the first-order fragment of OCL [13], that is the fragment of OCL that can be seen as fully declarative and encodable into relational algebra. Essentially, this excludes OCL operations involving *iterate*, *closure*, basic data type operations (such as String *concat*), and *OrderedSet* and *Bag* data types.

BPMN. BPMN (*Business Process Model and Notation*) [3] is a widely used and well-known ISO and OMG standard language for modeling business processes. It provides a graphical and intuitive notation which can be easily understood by business people, analysts and developers. In a nutshell, the language uses nodes to represent the activities or tasks of the process, whose execution order is determined by a set of directed edges. Different gateway nodes are available to control the flow, to allow for parallel or alternative execution paths, for instance. Moreover, using BPMN it is also possible to represent the interaction between different parties involved in the process, the message flow between them or the objects involved in the process, just to mention a few examples. The diagram has token semantics. As the different activities take place, the token (or tokens) flows through the diagram allowing the execution of the following activities. Due to this, it is possible to formalize a subset of the language into a Petri net [16]. This results in precise execution semantics for the BPMN diagram.

3   Linking Data and BPMN Models

We illustrate our proposal for linking process and data by means of the following example. As we are going to see, the main advantage of our proposal is that, in addition to the benefits of an artifact-centric approach which lets us represent both the structural (i.e., the data) and the dynamic (i.e., the activities or tasks) dimensions of the process, our models provide enough information to achieve their automatic executability.

Example. We aim at realizing a process to create playlists from tracks of musical albums. In particular, the process should deal with the following data and process flow:

- **Data:** Each album has a title, a date of first release and exactly one associated artist. An artist has a name and is either a physical person or a group. Each artist has one album at least. Albums contain one or more tracks. Each track has a number, a name and a duration and belongs to exactly one album. Some albums are special editions and, in that case, may contain bonus tracks. Playlists have a name and contain a nonempty set of tracks (for simplicity the order is not of interest).
- **Process flow:** Iteratively, the process asks the user for the name of an artist and continues with two parallel branches. The first calculates and returns to the user the set of tracks that are part of a special edition recorded by the artist; then, it asks the user to select a subset of these tracks and builds a playlist with them. In the second, the process obtains the set of playlists containing a track by the selected artist. At the end of the two branches, the set of tracks in the new playlist is returned to the user. After this, the user decides whether he/she wishes to continue adding playlists to the system or end the process.

In our proposal, we express the data requirements as a UML class diagram (see Figure 1), while the process flow is expressed in the BPMN (as shown in Figure 2). Notice that, as usual in BPMN, we have adopted *message events* for simple activities that only catch data from the user, or throw data to the user. These include `ArtistSelected`, `TracksPLnameSelected`, `PlaylistSent`, and `Continue`.

Now, our goal is to link the process events with the data. To do so, we need to ensure that the UML class diagram contemplates all the data modified/accessed in every atomic
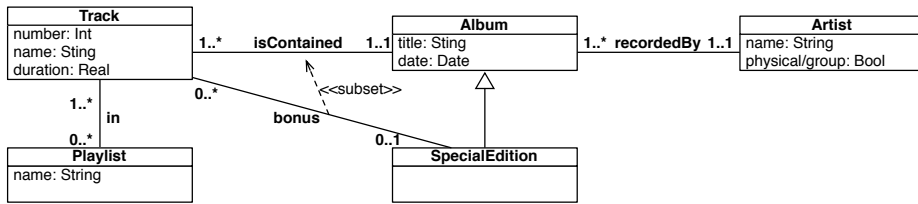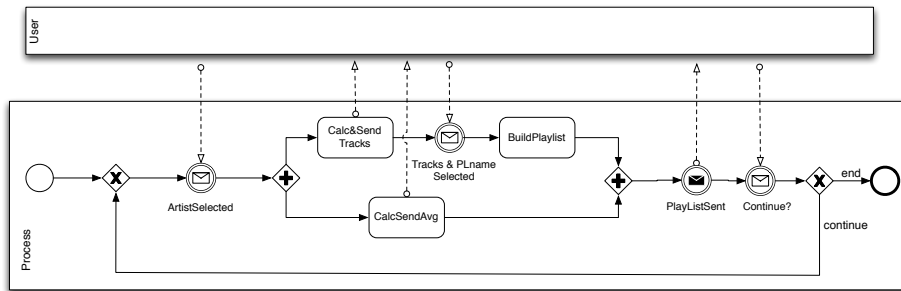
Fig. 1: Class diagram for our Playlist example

Fig. 2: BPMN diagram representing a process for creating playlists.

activity, decision, and message received or sent in the BPMN. Since, typically, the execution of a process needs to store some extra information in *process variables* (e.g., we need to remember the artist selected by the user at the beginning of the process since it is used in later BPMN events), we have to extend the class diagram to capture them. In particular, we consider a new class we call *Artifact* containing such *process variables*. To differentiate this class from the rest, we label it with the stereotype *Artifact*.

For instance, Figure 3 shows the Artifact for our ongoing example. This artifact is able to store the artist selected in the beginning of the process (through an association to *Artist*), the name of the playlist to create (through the attribute *plname*), the tracks to add in this new playlist (association to *Track*), the created playlist itself (association to *Playlist*), and whether the user selects to end the process or continue (attribute *end*).
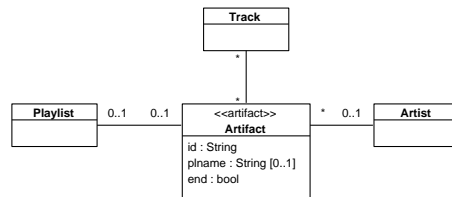

Fig. 3: Class diagram with the representation of the artifact

Representing the process variables as an *Artifact* class associated to the rest of elements in the class diagram provides the advantages of the object oriented paradigm. That is, we can specify modifications over the process data by specifying creations, deletions, and updates of objects/relations of that artifact. Note that, in this way:

– We avoid errors in the execution of the model, as we ensure that the artifact is linked to a specific instance of a class and not to an id of an instance which may not exist, due to the fact that the id is wrong.
– We simplify the definition of the operation contracts by manipulating objects (i.e. instances of classes) instead of identifiers.

Then, the idea is that, when a new process instance starts, a new *Artifact* object is created to store all these process variables. Observe that this behavior is similar to the use case controller in [17], as one class holds the required information for the execution of several related operations or tasks.

The UML class diagram and its instantiation, including the artifact, can be thought of as the *information model* of the process. Note that this instantiation can be seen (and in fact, stored) as a relational database (i.e., a first-order model).

Now, for any time instant, we define the *state* of the process as: a) The instantiation of the UML class diagram including the artifact; b) The positions of the tokens in the BPMN diagrams. Using this notion of state, we can describe precisely the process in terms of *state evolution*. For instance, our previous process can be described precisely as follows:

1. At the beginning of an iteration a message with the selected artist as payload comes in; such artist is stored in the Artifact through the corresponding association.
2. Then, concurrently the process follows two branches.
   – First branch:
     (a) The `CalcSendTracks` activity calculates all tracks that are part of some special edition recorded by the artist in `artist` and sends them to the user; the tracks resulting from the calculation are *not* stored in the Artifact, as they are not further used in the process, but are instead directly sent to the user.
     (b) Then, the user sends in the selected tracks and the name of the new play list. Both of these pieces of information are stored in the Artifact.
     (c) Using the Artifact stored `tracks` and `plname`, the `BuildPlayList` activity creates a new playlist. Such playlist is then stored in the Artifact.
   – Second branch:
     (a) The `CalcSendPlaylists` activity, starting from the Artifact's stored artist, collects all its tracks, computes the set of playlists that already exist which contain tracks by the selected artist and sends it to the user. Notice that, since this result is not used anymore in the process, it is *not* stored in the Artifact.
3. After these two branches complete their computations and join, a message with the newly created playlist is sent to the user.
4. Finally the `Continue?` activity gets the info of whether the user wants to continue or not, and stores it in the Artifact boolean variable `end`. Then, depending on this information, the XOR-gateway ends the process or performs another iteration.

This description of the state evolution can be made completely executable by (1) specifying the previous activities and start/end/message event through a formal language; and (2) adopting the Petri Net semantics for BPMN control flow.

Thus, for this purpose, we specify each activity in the BPMN diagram through an OCL operation contract. Each OCL operation will have a precondition, stating the conditions that must be true *before* the task can take place, and a postcondition, indicating the resulting state of the system *after* the operation's execution. Some of the tasks will only return information to the user without making any changes (we will call them queries): these tasks will include the keyword `result` as part of the postcondition. OCL operation contracts need to refer to the instances of `Artifact` to get rid explicitly of the information manipulated by the process.

In Table 1 we show the OCL operation contracts for the BPMN diagram in Figure 2. Note that we have also specified a contract for the start and end event in this diagram. The former (`Initialize`) is in charge of instantiating the artifact that will keep the information for the execution of the current process. The latter (`End`) is in charge of deleting the artifact and its relationships. Except for the task `Initialize`, which is a class operation, the rest of the tasks are instance operations invoked over the artifact being manipulated by the process (the one created by `Initialize`).

4   Achieving Executable Business Process Models

To make this framework executable, we encode the UML class diagram as a relational database manageable through SQL, the BPMN diagram as a Petri net, and the OCL contracts as logic rules that derive which SQL statements must be applied to the database when the corresponding activity is executed. In this way, we get the executability of the framework benefiting from standard relational database technology.

From the Class Diagram to a Database Schema. We encode the UML class diagram into a relational database following well-known techniques of database design [18]. Note that in this step we also store the Artifact (i.e., the process variables) in the database since the Artifact appears in the UML schema.

From the BPMN diagram to a Petri net. The BPMN diagram can be formalized into a Petri net by following [16]. This proposal focuses on formalizing the control-flow (i.e. the execution order of the tasks and events) of BPMN models, which is exactly what we need in this case. Roughly, each task will map to a transition with one input and one output place. Gateway nodes will, in the general case, correspond to a combination of places and silent transitions, to represent the routing behaviour of the gateway. This translation to a Petri Net is needed to make sure formally that the order of execution of the processes is exactly the one defined by the BPMN.

Petri nets also require an initial marking, which represents the initial state of the BPMN model. In general, this means placing a single token in the place that corresponds to the start node of the BPMN model. By following the token semantics of the resulting Petri net, it is possible to know exactly which tasks or events are ready to take place.

⚪ Start Event	**context** artifact::Initialize() **post:** Artifact.allInstances()->exists(af | af.oclIsNew() **and** af.end=false **and** result=af)
	Initialize creates a new artifact with its end attribute set to false.
✉ Artist Selected	**context** artifact::ArtistSelected(artist:Artist) **post:** self.artist=artist
	ArtistSelected assigns the artist given as input to the process's artifact.
activity Calc Send Tracks	**context** artifact::CalcAndSendTracks(): **Set**(Track) **post:** result = Track.allInstances()->select(t | t.album.artist = self.artist **and** t.album.oclIsTypeOf(SpecialEdition))
	CalcSendTracks obtains all the tracks and selects those belonging to an album whose artist is equal to the artist linked to the artifact and which are part of an special edition. It returns this list as a result.
✉ Tracks Plname Selected	**context** artifact::TracksPlnameSelected(trackL:**Set**(Track), plName:**String**) **post:** self.track=trackL **and** self.plname=plName
	TracksPlnameSelected assigns the set of tracks provided as input to the artifact, and stores the playlist name given as input in the corresponding attribute of the artifact.
activity Build Playlist	**context** artifact::BuildPlaylist() **post:** Playlist.allInstances()->exists(pl | pl.oclIsNew() **and** pl.name=self.plname **and** pl.track->includesAll(self.track))
	BuildPlaylist creates a new instance of Playlist (oclIsNew). Its name is the name stored in the artifact and its tracks will correspond to the tracks linked to the artifact.
activity Calc Send Playlists	**context** artifact::CalcSendPlaylists(): **Set**(Playlist) **post:** result = self.artist.album.track.playlist->asSet()
	CalcSendPlaylists obtains the playlists that already exist which contain tracks by the selected artist and sends this information to the user.
✉ Playlist Sent	**context** artifact::PlaylistSent(): Playlist **post:** result = self.playlist
	PlaylistSent returns the playlist that has been created (the one assigned to the artifact) as a result.
✉ Continue	**context** artifact::Continue(e:bool) **post:** self.end=e
	Continue updates the value of attribute end in the artifact with the given input.
⚪ End Event	**context** artifact::End() **post:** Artifact.allInstances()->excludes(self)
	End deletes the artifact linked to this instance of the process and all the relationships it takes part in.

Table 1: OCL contracts for the events and activities of the BPMN diagram

The Petri net we obtain in our example is shown in Figure 4. Each task corresponds to a labelled transition, which has one input and one output place. Each gateway node maps to a set of places and transitions. For instance, the XOR merge gateway placed before the task ArtistSelected corresponds to the transitions and places inside the dotted rectangle in Figure 4. The initial marking consists in putting a token in the most left-side place (the one with no input arcs).

From the OCL Operation Contracts to Logic Derivation Rules. Each OCL operation contract is encoded into a set of logic rules which, intuitively, derive the SQL insertions/deletions/updates that we must perform on the SQL database when apply-
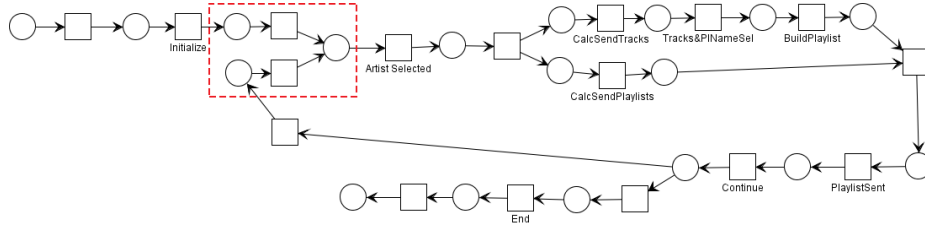
Fig. 4: Petri net resulting from the transition of the BPMN. The dotted rectangle shows the transitions and places corresponding to the translation of the XOR merge gateway placed before `ArtistSelected`.

ing the operation. In this way, we move from the declarative OCL specifications to an imperative formalism that can be executed.

The logic rules we obtain from each operation have the following form:

$$ins\_P(\overline{x}) :- opName(a), arg0\_opName(\overline{x_0}), ..., argN\_opName(\overline{x_n}), pre(\overline{x_{pre}}), query(\overline{x_q})$$
$$del\_P(\overline{x}) :- opName(a), arg0\_opName(\overline{x_0}), ..., argN\_opName(\overline{x_n}), pre(\overline{x_{pre}}), query(\overline{x_q})$$
$$result(\overline{x}) :- opName(a), arg0\_opName(\overline{x_0}), ..., argN\_opName(\overline{x_n}), pre(\overline{x_{pre}}), query(\overline{x_q})$$

The head of each rule determines the kind of SQL statement to apply (insertion, deletion, or query), while the body specifies for which values. That is, intuitively, a rule of the first form states that when a user invokes operation *opName* to artifact *a* with the n arguments specified in *arg0_opName*, ..., *argN_opName*, then some facts $P(\overline{x})$ must be inserted in the database if the precondition encoded by $pre(\overline{x_{pre}})$ is satisfied.

The variables $\overline{x}$ are instantiated using the arguments given by the user $\overline{x_0}$, ..., $\overline{x_n}$, or even the result of a first-order query *query($\overline{x_q}$)* that retrieves values from the current database state (or process data stored in the artifact a). If the query returns a set of tuples, or one argument itself is a set of tuples, the rule derives as many insertions as elements in the set.

Similarly, rules of the second and third form state deletions of facts and specify the tuples to return to the user as result. Attribute modifications are encoded by using the well-known strategy of combining a deletion and an insertion rule for the same fact.

The translation from OCL contracts to this logic formalism is an extension of the one in [19]. In particular, the extension we propose in this paper is intended to: (1) allow using the query *query($\overline{x_q}$)* to instantiate the variables used in the insertions/deletions to apply, (2) deal with *OCL Set* typed arguments, and (3) retrieving results for the user.

Given an OCL contract, its translation into logics consists in two steps. The first one parses the OCL postcondition to identify the different rules we need to create (i.e., it identifies the heads of the different rules to build). The second is in charge of creating the bodies of these rules, which is done by parsing the operation name, arguments, and the pre/postcondition to identify how to instantiate the variables from the rule head.

Identifying the head. We analyze the OCL postcondition to determine which kind of *updates* are performed by the operation. Essentially, such updates are: object cre-

ation/deletion, object specialization/generalization, relationship insertion/deletion, attribute insertion/deletion/modification, and queries. Each of these updates will lead to one or several derivation rules. For instance, an object creation of class C, where C is a subclass of C', leads to a derivation rule of the form $ins\_C(o)$, together with another derivation rule of the form $ins\_C'(o)$. Intuitively, the set of derivation rules generated for each object insertion/deletion performs the *ISA closure* as stated in the Preliminaries.

In Table 2 we show how we identify such updates and the derivation rules they originate. This table is an extension of the translating rules defined in [19] to sets and data extracted from the database. Intuitively, we traverse the OCL postcondition to find the OCL patterns stated in the left column of the table and, for each match, we create a new derivation rule as stated in the right column. In this table, we use *o* and *u* to refer to OCL object expressions of type C, and *a* and *b* to refer to OCL value expressions (such as constants). Moreover, we use *role* to refer to property call navigations through associations R, *attr* to property call navigations to attributes A, and *query* to refer to an OCL query expression. Finally, we assume that $\bar{t}$ is a tuple of n variables, where n is the arity of the *TupleType* returned by the OCL query, or 1 if the OCL query returns an object/value.

Table 2: OCL patterns to derivation rules

OCL pattern	Update kind	Derivation Rules to Create
o.oclIsNew()	Object creation	ins_C(o) ins_C'(o), for each C $\sqsubseteq$ C'
C.allInstances()->excludes(o)	Object deletion	del_C(o) del_C'(o) for each C' $\sqsubseteq$ C del_C"(o) for each C $\sqsubseteq$ C"
o.oclIsKindOf(C')	Object specialization	ins_C'(o) ins_C"(o) for each C' $\sqsubseteq$ C" $\sqsubseteq$ C
not o.oclIsKindOf(C')	Object generalization	del_C'(o) del_C"(o) for each C" $\sqsubseteq$ C'
o.role->includes(u) o.role->includesAll(u)	Relationship insertion	ins_R(o, u)
o.role->excludes(u) o.role->excludesAll(u)	Relationship deletion	del_R(o, u)
o.oclIsNew() and o.attr = a	Attribute insertion	ins_A(o, a)
o.attr = null	Attribute deletion	del_A(o, a)
o.attr = b	Attribute update	ins_A(o, b) del_A(o, a)
result = query	Query	result($\bar{t}$)

Deriving the body. Once we know the kind of updates each operation applies, we have to determine the values for which they should be applied. This is achieved by means of the expression in the body of the rule, which consists of two different parts: one which is common to all derivation rules of each operation specifying the operation name, arguments and precondition; and a specific part for each derivation stating the specific

queries (i.e., a conjunctions of literals referring to the database state) used to instantiate the variables in the rule. We explain each part in the following.

– *Common part of the body.* The common part of the body consists of one literal representing the operation we are translating *opName(a)*, whose unique variable represents the artifact in which we are applying the operation, the arguments *arg0_opName($\overline{X_0}$), ... , argN_opName($\overline{X_n}$)* representing the values given by the user to perform such operation, and one logic query *pre($\overline{X_{pre}}$)* encoding the precondition of the operation. Such logic query is obtained by translating the OCL precondition into logics according to the proposal in [20].
– *Specific part of the body.* The queries in this part are obtained through the logic translation of the *o*, *u* object expressions, *a*, *b* value expressions and the *query* expression appearing in Table 2, which is only performed if the expressions do not explicitly refer to operation arguments (since they have been encoded already previous step). We also use [20] to perform this encoding. Essentially, this consists in translating each OCL navigation as a sequence of logic atoms representing the different associations it traverses to. For instance, *t.album.artist* is translated into *track(T, Al) $\wedge$ recordedBy(Al, Ar)*. The idea of the translation is that, each logic variable used in the navigations represents a different UML object, and thus, can be further used to state conditions over such object. For instance, *specialEdition(Al)* states that the UML object represented by the variable *Al* is a specialEdition.

As an example, the OCL contracts of the operations *BuildPlaylist* and *CalcSend-Tracks* are translated as the rules set shown in Listings 1.1, and 1.2. Note that the variables in the head of the rules are instantiated using queries in the body of the rules.

Listing 1.1: Logic encoding for task *BuildPlaylist*

```
ins_Playlist(Pl) :- buildplaylist(A), artifactPlname(A,Pl)
ins_TrackIn(Tr, Al, Pl) :- buildplaylist(A), artifactPlname(A,Pl),
    artifactTrack(A,Tr,Al)
ins_ArtifactPlaylist(A,Pl) :- buildplaylist(A), artifactPlname(A,Pl)
```

Listing 1.2: Logic encoding for task *CalcSendTracks*

```
result_CalcSendTracks(T,Al) :- calcSendTracks(A), track(T,Al), recordedBy(Al,
    Ar), artifactArtist(A,Ar), specialEdition(Al)
```

5   Executing the framework

The proposed framework allows us to automatically and unambiguously execute processes defined according to our specification models. We have built a Java library for this purpose [4]. This library permits loading in compilation time the underlying semantic models of the framework and executing its operations at runtime. That is:

---

[4] A prototype of this library together the necessary code/models to execute the BPM used in this paper can be found at `http://www.essi.upc.edu/~xoriol/opexec/`.

- *Given (at compilation time)*: (1) a SQL database connection encoding a UML schema, (2) a set of derivation rules defining the semantics of the operations, (3) a map from the logic predicates to the SQL tables/columns.
- *Given (at runtime)*: (4) an operation name, (5) the values for their arguments.
- *Executes (at runtime)*: (6) the updates specified in the derivation rules of the operation in the database, and (7) returns to the user the information specified in the *result* part of the operation.

The current version of the Java library does not check yet whether the operations executed by the user match the order imposed by the Petri nets. However, we understand that this critical (and necessary) functionality may be achieved by integrating any Petri Net simulator in our library and this is why we have left implementation of this part for future work. In contrast, the tool works in any relational database management system and it is able to check whether the executed operations cause the violation of some integrity constraint (such as the min/max multiplicity constraints of the UML class diagram, other UML class diagram annotations such as *subset*), by means of the implementation of the incremental integrity checking approach in [21].

Operation Executor Library Architecture. The architecture of our library is shown in Figure 5. Briefly, a user loads (at compilation time) the previous models in the *Controller* component, which stores them. When the user wants to start executing the process, he/she invokes the controller to instantiate a new *ProcessExecutor*. This class executes all the operation invocations of such process instance. Thus, each instance of this class has its own (unique) artifact ID, which is used to store, in the database, all the process data related to such process instance. When a user invokes an operation to the *ProcessExecutor*, the *ProcessExecutor* creates an *OperationExecutorThread*, in which we store the derivation rules related to such operation. Then, the *ProcessExecutor* adds it to the *OperationExecutionThreadManager*. This component is in charge of executing the operation of this operation as soon as it can be executed. When the *OperationExecutionThread* is executed, it performs the following steps:

1. It instantiates the updates (insertions/deletions) that it must apply according to the derivation rules and the database state.
2. It checks that these updates do not cause any constraint violation according to the incrementally checking method defined in [21].
3. If no violation is found, the updates are translated as SQL insert/delete/update statements and executed, and the query to retrieve the result of the operation execution is performed (if the operation returns some result).
4. Otherwise, an exception is thrown.

6   Related Work

In the following, we first discuss related frameworks for linking data and process models, and then, discuss several of their formalizations to achieve their executability.
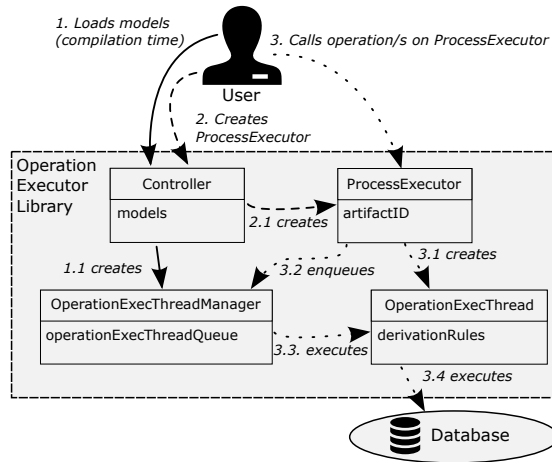
Fig. 5: OperationExecutor Java Library Architecture

In terms of the framework for modeling data and business processes, many of the existing works [9–11] use languages grounded on logic, which are formal and unambiguous but more difficult to understand than BPMN and UML. There are other approaches which use graphical representations which are more intuitive and appealing to business analysts and developers, such as [12,22,23]. [23] is based on the Guard-Stage-Milestone approach, which represents the evolution of each relevant object in a lifecycle following a more declarative approach than ours. [22] uses *artifact union graphs*, which are similar to Petri nets, to represent the process. [12] is the most similar approach to ours and relies on various UML diagrams (different to the ones we consider) and OCL contracts to represent the data and the process. However, none of these works deal with process executability; most of them focus on studying the correctness of the model.

Regarding process executability, BPEL (or WS-BPEL) allows to specify executable business processes using an XML format which makes it difficult to read. Although there is a mapping between BPMN 2.0 and BPEL it is incomplete and suffers from several issues [24]. The work of [25] uses XML nets, a Petri-net-based process modelling approach which is meant to be executable. It uses a graphical language, which maps to a DTD (XML Document Type Definition) to represent the data required by the process, and the data manipulations are graphically shown in the XML net. In contrast to our approach, this solution is technology-based, as the specification of the models is based on XML, and details of how to achieve executability are not explained.

YAWL [26] is a workflow graphical language whose semantics are formally defined and based on Petri nets, with its corresponding execution engine. The language offers both a control-flow and data-flow perspective of the process, where data is defined following an XML format. Intuitively, the tasks are then annotated with their inputs and outputs, but they do not allow defining what changes are made by each of them. Therefore, the execution engine only detects missing information and it is not able to fully execute the operation.

In [27] an hybrid model using a data-oriented declarative specification and a control-flow-oriented imperative specification of a business process are defined. Using this ap-

proach it is possible to obtain automatically an imperative model that is executable in a standard Business Process Management System. However, data is defined as a set of unstructured variables and the pre and postconditions merely state conditions over the data, instead of indicating exactly what is done by the different tasks.

Earlier, similar attempts to ours are [28, 29]. Both approaches focus on defining a conceptual model which can then be automatically translated to achieve execution. However, the purpose of [28] is different to ours: their main goal is to be able to validate the model through execution, while ours is to achieve executability by using the current *de facto* standard languages for data and process representation. Similarly, the approach in [29] - which translates the models into Pascal - is outdated by current, object-oriented programming languages.

In addition, it is worth noting that most of these proposals do not use standard formalisms for conceptual representation, as we do.

7   Conclusions

We have proposed a framework to link data and business processes, which can be to be executed automatically. It uses the BPMN language to represent the processes, the UML class diagram for the data, and OCL operation contracts to define what do the tasks in the process. Using these languages, we are not proposing any yet-another-formalism but using a standard one in a new integrated way to link data and processes.

We have shown the feasibility of our approach by creating a Java library which, given a model, is able to execute the tasks and update the information base accordingly. Before applying the changes, the tool performs an incremental checking of integrity constraints to determine if there are any violations. If this is the case, it will throw an exception. Otherwise, it applies the changes to the underlying database that stores the data. All of this is performed without requiring user intervention.

With the approach we present here, we blur the distinction between specification and implementation, since the specification itself is executable.

References

1. OMG: Unified Modeling Language (UML) superstructure, version 2.0. Available at `http://www.uml.org/` (2005)
2. Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer (2007)
3. Dumas, M., Rosa, M.L., Mendling, J., Reijers, H.: Fundamentals of Business Process Management. Springer (2013)
4. Reichert, M.: Process and data: Two sides of the same coin? In: Proc. of the On the Move Confederated Int. Conf. (OTM 2012). Volume 7565 of LNCS., Springer (2012) 2–19
5. van der Aalst, W.M.P.: A decade of business process management conferences: Personal reflections on a developing discipline. In: Proceedings of BPM-12. (2012)

6. Cohn, D., Hull, R.: Business artifacts: A data-centric approach to modeling business operations and processes. IEEE-BDE **32**(3) (2009) 3–9
7. Bhattacharya, K., Caswell, N.S., Kumaran, S., Nigam, A., Wu, F.Y.: Artifact-centered operational modeling: Lessons from customer engagements. IBM Journal **46**(4) (2007) 703–721
8. Hull, R.: Artifact-centric business process models: Brief survey of research results and challenges. In: OTM Confederated Int. Conf. (2008)
9. Deutsch, A., Hull, R., Patrizi, F., Vianu, V.: Automatic verification of data-centric business processes. In: Proc. of ICDT. (2009) 252–267
10. Bagheri Hariri, B., Calvanese, D., De Giacomo, G., Deutsch, A., Montali, M.: Verification of relational data-centric dynamic systems with external services. In: Proc. of PODS. (2013) 163–174
11. Belardinelli, F., Lomuscio, A., Patrizi, F.: Verification of agent-based artifact systems. J. Artif. Intell. Res. **51** (2014) 333–376
12. Estañol, M., Sancho, M., Teniente, E.: Verification and validation of UML artifact-centric business process models. In: CAiSE 2015. Volume 9097 of LNCS., Springer (2015) 434–449
13. Franconi, E., Mosca, A., Oriol, X., Rull, G., Teniente, E.: Logic foundations of the OCL modelling language. In: Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings. (2014) 657–664
14. Fowler, M., Scott, K.: UML Distilled – Applying the Standard Object Modeling Laguage. Addison-Wesley (1997)
15. OMG: Object Constraint Language (UML), version 2.4. Object Management Group (OMG). (2014) `http://www.omg.org/spec/OCL/`.
16. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Information & Software Technology **50**(12) (2008) 1281–1294
17. Larman, C.: Applying UML and Patterns. 2nd edition edn. Prentice Hall (2002)
18. Teorey, T., Lightstone, S., Nadeau, T.: Database Modeling and Design. Fourth edn. Morgan Kaufmann, San Francisco (2006)
19. Queralt, A., Teniente, E.: Reasoning on UML conceptual schemas with operations. In: CAiSE 2009. Volume 5565 of LNCS., Springer (2009) 47–62
20. Queralt, A., Teniente, E.: Verification and validation of UML conceptual schemas with OCL constraints. ACM Trans. Softw. Eng. Methodol. **21**(2) (2012) 13
21. Oriol, X., Teniente, E.: Incremental checking of OCL constraints with aggregates through SQL. In: 34th International Conference on Conceptual Modeling, ER 2015. (2015) 199–213
22. Borrego, D., Gasca, R.M., López, M.T.G.: Automating correctness verification of artifact-centric business process models. Information & Software Technology **62** (2015) 187–197
23. Damaggio, E., Hull, R., Vaculín, R.: On the equivalence of incremental and fixpoint semantics for business artifacts with Guard-Stage-Milestone lifecycles. Inf. Syst. **38**(4) (2013) 561–584
24. Fabra, J., de Castro, V., Álvarez, P., Marcos, E.: Automatic execution of business process models: Exploiting the benefits of model-driven engineering approaches. Journal of Systems and Software **85**(3) (2012) 607–625
25. Lenz, K., Oberweis, A.: Modeling interorganizational workflows with XML nets. In: HICSS-34, IEEE Computer Society (2001)
26. Foundation, T.Y.: YAWL - User Manual. Version 4.1. (2016) Available at: `http://www.yawlfoundation.org/pages/support/manuals.html`.
27. Parody, L., López, M.T.G., Gasca, R.M.: Hybrid business process modeling for the optimization of outcome data. Information & Software Technology **70** (2016) 140–154
28. Lindland, O.I., Krogstie, J.: Validating conceptual models by transformational prototyping. In: CAiSE'93. Volume 685 of LNCS., Springer (1993) 165–183
29. Mylopoulos, J., Borgida, A., Greenspan, S.J., Wong, H.K.T.: Information system design at the conceptual level - the taxis project. IEEE Database Eng. Bull. **7**(4) (1984) 4–9