Specifying Artifact-Centric Business Process Models in UML: Technical Report

Montserrat Estañol[1], Anna Queralt[2], Maria-Ribera Sancho[1,2], and Ernest Teniente[1]

[1] Universitat Politècnica de Catalunya, Barcelona, Spain
estanyol|ribera|teniente@essi.upc.edu,
[2] Barcelona Supercomputing Center, Barcelona, Spain
anna.queralt@bsc.es

Abstract. In recent years, the artifact-centric approach to process modeling has attracted a lot of attention. One of the research lines in this area is finding a suitable way to represent the dimensions in this approach. Bearing this in mind, this paper proposes a way to specify artifact-centric business process models by means of well-known UML diagrams, from a high-level of abstraction and with a technology-independent perspective. UML is a graphical language, widely used and with a precise semantics.

Key words: Business Artifacts, BALSA Framework, UML, Business Process Modeling

1 Introduction

Business process modeling (BPM) is one of the most critical tasks in the business's definition, as business processes are directly involved in the achievement of an organization's goals, and thus they are key to its success. When modeling business processes, it is important that the final models are understandable by the people involved in them. Moreover, they should be formal and precise enough in order to be able to automatically check their correctness at definition time, thus preventing the occurrence of errors when the business is deployed.

Traditionally, business processes have been modeled following a process-centric approach, which focuses on the activities or tasks in the process, undermining the data needed to carry them out. In contrast, in the artifact-centric approach the data required by the processes plays a key role in their definition. In particular, business artifacts model key business-relevant entities which are updated by a set of services that implement the business process tasks.

In addition to *business artifacts*, an artifact-centric approach to process modeling should include a way to specify the *lifecycle* of the artifacts, i.e. the relevant stages in their evolution; their *associations*, i.e. the conditions under which changes are made to the artifacts and the *services* that are in charge of evolving them. By using different models and constructs in each of these dimensions, one can obtain different process models with diverse characteristics. One of the research lines in this area is focused on finding a suitable way of representing these dimensions.

The artifact-centric approach has great intuitive appeal to business managers and developers [1] and it has been successfully applied in practice [2]. An additional advantage of this approach over the process-centric one is that the presence of data in the models facilitates performing automated reasoning on them. That is, it is possible to define formally what each task does and to assess whether the models are correct considering the meaning of the tasks and the requirements of the business.

Following these ideas, we propose to specify artifact-centric business process models by means of well-known UML diagrams, from a high-level of abstraction and with a technology-independent perspective. UML is a graphical language, widely used and with a precise semantics. Therefore, it may be understandable by people involved in the business process, both from the business and from the system development perspectives. UML provides also extensibility mechanisms that permit more flexibility without losing its formality. These characteristics are important requirements in artifact-centric process modeling [3].

Generally, UML diagrams make use of some textual notation to precisely specify those aspects that cannot be graphically represented. We will use the OCL (Object Constraint Language) for that purpose.

The choice of using UML diagrams does not necessarily restrict our approach to this language since alternative diagrams or languages could be used for modeling some of the dimensions, provided that they allow specifying all the features required on it. We have chosen UML for the advantages just mentioned and because it intuitively maps to the dimensions.

Currently, several alternatives have been proposed to model artifact-centric business processes, such as Guard-Stage-Milestone (GSM) models [4, 5, 6], BPMN with data [7] or PHILharmonic Flows [8], to mention a few examples. However, as we will see, these approaches either do not use the same language to represent all the dimensions or the chosen representation is not graphical - it is often based on some variant of logic - making the models difficult to understand. The use of natural language in some of the proposals may lead to ambiguities and errors.

Our approach allows also automated reasoning from the business process models (as shown on [9, 10]), while most of the existing proposals that handle reasoning are based on models which use languages grounded on complex mathematical notations [11, 12, 13] which are not practical at the business level.

The work we present in this paper extends our work in [14, 15] by presenting a detailed methodology to model business processes from an artifact-centric perspective. We illustrate this methodology by means of a complex example, taken from [16], which requires handling multiple business artifacts interacting together (and not only single-artifact systems as considered in our previous work). We also outline the different alternative diagrams that might be used for modeling each dimension and provide a more detailed comparison with related work.

2 Our Approach to Artifact-centric Process Modeling

The artifact-centric approach to business process modeling provides four explicit, inter-related but separable, dimensions in the specification of the business process, as described in the BALSA framework [1]: Business Artifacts, Lifecycles, Associations and Services. We summarize here the most relevant characteristics of each dimension:

- **Business artifacts** represent the data required by the business and whose evolution we wish to track. Each artifact has an identifier and may be related to other artifacts, as represented by the associations among them.
- The **lifecycle** of a business artifact states the relevant stages in the evolution of the artifact, from the moment it is created until it is destroyed. Each business artifact is going to have a lifecycle.
- **Associations** establish the conditions under which the activities of the business process should be executed. That is, they determine the execution order of the services to allow the artifact to perform a transition from one stage of its lifecycle to another.
- **Services**, or tasks, represent atomic units of work and they are in charge of creating, updating and deleting the business artifacts. They correspond to the atomic acitivites of the associations, i.e. those which are not further decomposed.

Apart from business artifacts, businesses may also need to store data that does not really evolve. We will refer to this data as *objects*.

The modeling approach we propose here is based on representing the BALSA dimensions using UML and OCL: UML class diagrams for business artifacts; UML state machine diagrams for lifecycles; UML activity diagrams for associations, and OCL operation contracts for services. However, this choice does not restrict our approach to this subset of diagrams since, as we shall see, other alternatives may be used provided that they follow the methodology described. We call our approach *BAUML* (BALSA UML, for short).

Figure 1 shows the dimensions in the BALSA framework and their representation in the BAUML approach. Roughly, our methodology behaves as follows. Business artifacts correspond to some of the classes in the class diagram. For each artifact, a state machine diagram is defined stating its lifecycle. Then, each transition of the state machine diagram is further specified by means of an activity diagram determining the associations of the artifact. Finally, the behavior of the atomic activities from each activity diagram is precisely defined through an operation contract.

The remainder of this section presents in more detail our methodology for artifact-centric business process modeling using the BAUML approach. We also describe the components of the different diagrams and how they relate to the other diagrams.

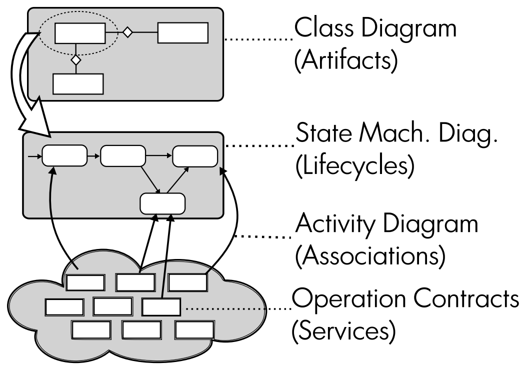


Fig. 1. Representation of the BALSA dimensions in our approach, adapted from [1]
.

2.1 Business Artifacts as a Class Diagram

The class diagram will have a set of classes and associations representing the data and their relationships as required by the business process. Some of the classes, those with an important dynamic behavior, will represent business artifacts. An artifact must necessarily be the top class of a hierarchy whose leaves are dynamic subclasses so that the artifact can change its type from one subclass to another.

Each subclass represents one of the specific states in the evolution of the artifact. They must fulfill the disjointness constraint (since an artifact cannot be in two states at the same time), but they can fulfill the completeness constraint (i.e. the artifact must have any of its subtypes) or not. If the artifact has a multi-level hierarchy, these rules apply to all the levels.

The advantage of using a hierarchy of subclasses to represent the potential states of an artifact is that it is possible to represent the attributes and relationships that are needed in each of the possible states while keeping the artifact's original identifier and the relationships that are common to all states (or several substates).

UML class diagrams can represent, in a graphical way, the classes with their corresponding attributes, the relationships between those classes, and integrity constraints. Artifacts will have stereotype `«artifact»` in their corresponding class. We will refer to the classes that do not correspond to business artifacts as objects.

Integrity constraints correspond to restrictions over the classes, the attributes, or the relationships between them. Those integrity constraints that cannot be represented graphically in the class diagram should be described in OCL to ensure their formality. However, they could also be specified using natural language for easier readability.

Alternative representations to the class diagram could be an ER or an ORM diagram. Both diagrams also allow defining the artifacts, the objects and their relationships in a graphical way.

2.2 Lifecycles as State Machine Diagrams

Each artifact in the class diagram will have a state machine diagram. This state machine diagram will have a set of states, a set of events, a set of effects and a set of transitions between pairs of states.

The states in the state machine diagram will correspond to the subclasses of the artifact if the hierarchy is complete. If it is incomplete, then the state machine diagram will have another state for the superclass. In this context, this state will represent an artifact that does not have any of the subtypes of the superclass. These rules apply to any multi-level hierarchy in the artifact.

The state machine diagram will also show the allowed transitions between states. Finally, we also define the initial states as a subset of the states which act as a target state for the initial transitions. Those initial transitions will always result in the creation of a new artifact instance.

Each transition will have a source state and a target state. Moreover, it may also have an OCL condition over the class diagram, an event and a tag representing the result from the execution of the event. We differentiate between three types of transitions (the elements inside parenthesis are optional):

- `([OCL]) ExternalEvent ([tag])`
- `([OCL]) TimeEvent (/ Effect)`
- `[OCL] (/ Effect)`

The first transition type will take place when `ExternalEvent` takes place and the OCL condition is true. If there is a `tag`, then the result of the execution of `ExternalEvent` must coincide with `tag` for the transition to take place. The second transition will take place when there is a `TimeEvent` and the OCL condition is true. If there is an `Effect`, the changes specified by it will also be made. Finally, the last transition type is similar to the second excepting the occurrence of a time event. These transition types cover the types of transitions allowed in the UML 2.4.1 specification that are significant at the specification level, as explained in [17].

An `ExternalEvent` will have as input parameters the artifacts in whose transitions it appears or the identifiers of those artifacts. The execution of these events and their respective `tag`s (if any) will be defined in an activity diagram. `Effect`s correspond to atomic tasks that have as input parameters the artifacts involved in the transition.

OCL is an OCL expression which starts from `self` or `Class.allInstances()->...` where `Class` is any of the classes in the class diagram. A `TimeEvent` represents an occurrence of time. We distinguish between relative and absolute time expressions. An absolute expression has the form `at(time_expression)`; a relative expression has the form `after(time_expression)`.

Notice that this state machine diagram does not follow exactly the UML standard described in [18]. This is due to the fact that it has tags, which we use to determine whether the event ends successfully or not. In traditional UML state machine diagrams, events are atomic and there is no need for such conditions.

In addition, we also allow more than one outgoing transition from the initial node. This is useful when the artifact can be created in different ways. Alternatively, this situation could be represented using one outgoing transition from the initial node, leading to a state called *InitialState*. From this state, we could have the outgoing transitions that start from the initial node and leave the rest of the state machine diagram as it is. However, representing the lifecycles in this way does not contribute any relevant information and adds complexity to the final diagram.

Although we use a variant of UML state machines, any other notation based on state machines would be useful to represent the lifecycles of the artifacts.

2.3 Associations as Activity Diagrams

For every `ExternalEvent` in a state machine diagram, there will exist exactly one activity diagram. An activity diagram will have a set of nodes and a set of transitions between those nodes. More specifically, the activity diagram will have exactly one initial node and one or several final nodes. Transitions will determine the change from one node to the next. Apart from a source node and a target node, transitions may also have a `guard condition` and a `tag`. The `tag` will determine the correct or incorrect execution of the activity diagram, and will connect it to the right transition in the state machine diagram.

We distinguish between the following node types:

– **Initial Node:** Point where the activity diagram begins
– **Final Node:** Point where the flow of the activity diagram ends.
– **Gateway Node:** Gateway nodes are used to control the execution flow. We distinguish between *decision nodes*, *merge nodes*, *inclusive-or nodes*, *fork nodes* and *join nodes*.
– **Activity:** An activity represents work that is carried out. We differentiate three types of activities. A *task* corresponds to a unit of work with an associated operation contract. The operation contract will have a precondition, stating the conditions that must be true for the task to execute, and a postcondition, indicating the state of the system after the task's execution. Both are formalized using OCL queries over the class diagram. *Material actions* correspond to physical work which is carried out in the process but that does not alter the system. Finally, a *subprocess* represents a "call" to another activity diagram, and as such may include several tasks and material actions.

We assume the following: decision nodes and fork nodes have one incoming flow and more than one outgoing flow; merge nodes and join nodes have several incoming flows and exactly one outgoing flow; activities have one incoming flow and one outgoing flow; initial flows have no incoming and one outgoing flow; and final nodes may have several incoming flows but no outgoing flow.

Guard conditions are only allowed over transitions which have a decision or an inclusive-or node as their source. The guard condition may refer to either:

– The result of the previous task

– An OCL condition over the class diagram
– A user-made decision

On the other hand, tags are only allowed over those transitions that have as target a final node.

During the execution of the activity diagram we assume that the constraints established by the class diagram may be violated. However, at the end of the execution they must be fulfilled, otherwise the transition does not take place and the changes are rolled back.

Finally, activity diagrams may also represent the main artifact involved in each of the tasks and its participants (i.e. the role of the person who carries out a particular activity) using swimlanes and notes, respectively as described in [14, 15]. However, for easier readability of the diagrams we do not show them in this paper.

Although we adopt the UML activity diagrams to represent the associations, they could also be represented using other notations (as long as they follow the semantics) such as BPMN or DFDs. BPMN is probably the language that is most used to represent business process models, and as such it offers a great variety of syntactic sugar for the basic node types described above. Data-Flow diagrams (DFD) are also another alternative, as they show the task and the inputs and outputs of data required and generated by them.

2.4 Tasks (Services) as Operation Contracts

As we have mentioned, each of the tasks in the activity diagrams will have an associated operation contract. The same applies to effects in the state machine diagrams. The contract will have a set of input parameters, a precondition, a postcondition and may have an output parameter. The input and output parameters may be classes or simple types (e.g. strings, integers). If several tasks belong to the same activity diagram and their input parameters have the same names, we assume that their value does not change from one task to the next.

The task can only be executed when the precondition is met, and the postcondition specifies the state of the system after the execution of the operation. We also assume a strict interpretation of operation contracts to avoid redundancies [19]. Those classes that do not appear in the postcondition keep their state from before its execution.

We choose OCL to represent the operation contracts because it is a formal language that avoids ambiguities, it integrates naturally with UML and is independent from the final implementation of the process. For easier readability, they could be specified in natural language, although we do not recommend it because it is prone to ambiguities and errors.

3 Running Example

Our running example is based on the backend process of a CD online shop, extracted from [16] and remodeled following our approach. The shop splits customer requests into different orders to the CD suppliers. The difficulty in this example lies in the representation of the relationship and the interaction between two different business artifacts: the quote requests made by the customers and the orders into which a quote request may be split (which in turn may involve several quote requests).

In particular, this shop keeps no stock of items, rather, the store obtains the CDs from its supplier after a customer request. The customer places a quote request for one or more CDs. Then the CD shop calculates the price of the order and informs the customer. If the customer accepts the quoted price, then the shop orders the CD to its suppliers, grouping in a single order to a supplier several quote requests. When the company receives the orders from the suppliers, they are then regrouped into the orders for the customers. The CD shop keeps track of the evolution of the quote requests from the customers and the orders the company makes to its suppliers.

3.1 Class Diagram

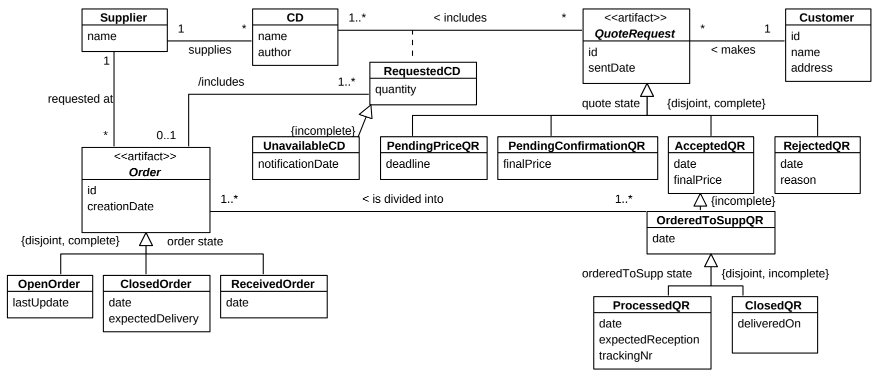
Figure 2 shows the class diagram for our example. There are two business artifacts: `QuoteRequest` and `Order`, as shown by the stereotypes. The rest of the classes in the diagram, such as `Supplier`, `Customer` or `CD` represent objects: relevant information for the business but whose evolution we do not track. Each artifact has its own identifier, in this case, for both `QuoteRequest` and `Order` the identifier is `id`. The rest of classes in the diagram may also have their identifiers, for instance, a `CD` is identified by both its `name` and `author`. Each artifact and object has as many attributes and relationships as relevant for the business[1].

Artifact `Order` is the simpler of the two. It has three different subclasses: `OpenOrder`, `ClosedOrder` and `ReceivedOrder` which contain the relevant information for that particular state of the artifact. An `OpenOrder` is waiting to be sent to the supplier and additional `QuoteRequest`s can be assigned to it. A `ClosedOrder` has already been sent to the supplier. Finally, an `Order` changes its state to `ReceivedOrder` when it has been received at the shop.

On the other hand, artifact `QuoteRequest` has a first level of subclasses which are `PendingPriceQR`, `PendingConfirmationQR`, `AcceptedQR` and `RejectedQR`. A `PendingPriceQR` is waiting for the shop to quote the price. A `PendingConfirmationQR` has already a price and is waiting for the customer's acceptance or rejection. An `AcceptedQR` has already been accepted by the customer. In contrast, `RejectedQR` has been rejected.

`AcceptedQR` has one subclass: `OrderedToSuppQR`. Notice that the hierarchy is incomplete. An `OrderedToSuppQR` has already been split into several `Order`s that will eventually be processed and sent to the suppliers. At the same time, `OrderedToSuppQR` has two subclasses: `ProcessedQR` and `ClosedQR`, and the hierarchy is incomplete. Like in the previous case, an `OrderedToSuppQR` may not

[1] Notice that we have not included the attribute types in the class diagram. This helps keeping it more compact. The types can be inferred from the OCL operation contracts.



1. Identifiers: (QuoteRequest, id), (Customer, id), (CD, name+author), (Supplier, name), (Order, id)
2. A *ProcessedQR* may not have more than one *Order* for the same *Supplier*.
3. Derived Relationship *includes*: A *RequestedCD* will be related to an *Order* if the *QuoteRequest* of the *RequestedCD* is linked to an *Order* requested at its CD *Supplier*.
4. There can only be at most one *OpenOrder* per *Supplier*.
5. The *Orders* into which a *ProcessedQR* is divided must be to *Suppliers* who provide the *CD*s included in the *ProcessedQR*.
6. The *sentDate* of a *QuoteRequest* must be earlier than attribute *date* in any of the subclasses of *QuoteRequest*.
7. $QuoteRequest.sentDate < PendingPriceQR.deadline$
8. $AcceptedQR.date \leq OrderedToSuppQR.date$
9. $OrderedToSuppQR.date \leq ProcessedQR.date$
10. $ProcessedQR.date \leq ClosedQR.deliveredOn$
11. The *creation* date of *Order* must be earlier or equal to attribute *date* in any of its subclasses.
12. $Order.creationDate \leq OpenOrder.lastUpdate$

Fig. 2. Class diagram showing the business artifacts as classes with the corresponding integrity constraints

have any of the subtypes. `ProcessedQR` represents a quote request that has already been sent to the customer, and a `ClosedQR` corresponds to a quote request that has already been received by him or her.

Notice the case of class `RequestedCD`. It is an association class that results from the reification of the relationship between `CD` and `QuoteRequest`, which allows us to record additional information about the relationship between two or more classes. In this case, `RequestedCD` is identified by `CD` and `QuoteRequest`. That is, association classes are identified by the classes that partake in the relationship.

3.2 State Machine Diagrams

Figures 3 and 4 show the state machine diagrams that correspond to the business artifacts in this example: `Order` and `QuoteRequest`. We will begin by looking at the state machine diagram for `Order`, which is simpler. In this case, there

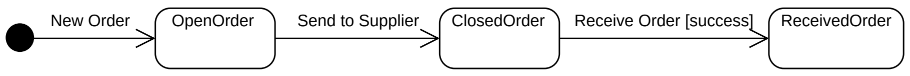


Fig. 3. State machine diagram for artifact `Order`.

is a single-level hierarchy in the class diagram with restrictions `disjoint` and `complete`, therefore the states exactly map to the subclasses in the class diagram. An `Order` is created when there is a request to create a new order, as shown by event `New Order`. This order remains in state `OpenOrder` until someone decides that the order can be made to supplier, by executing event `Send to Supplier`. Then the order becomes a `ClosedOrder` and no more `AcceptedQR`s can be linked to it. Finally, once the order is received, if event `Receive Order` executes successfully, as indicated by tag `success`, it changes its state to `ReceivedOrder`.

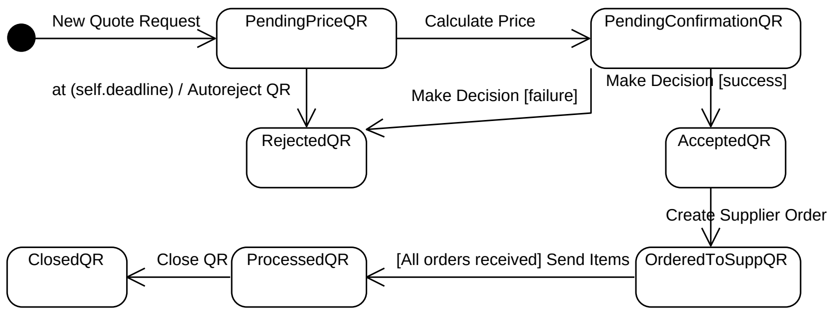


Fig. 4. State machine diagram for artifact `QuoteRequest`.

On the other hand, artifact `QuoteRequest` has a more complex state machine diagram. First of all, it has a multi-level hierarchy. More specifically, it has three different levels. The first level has constraints `disjoint` and `complete`, but the second and third levels are `incomplete`. In the first-level hierarchy, the states are: `PendingPriceQR`, `PendingConfirmationQR`, `RejectedQR` and `AcceptedQR`. Although `AcceptedQR` has two subclasses, it is included because the hierarchy is incomplete, and therefore, there can exist an `AcceptedQR` which has no subtypes.

When a customer wishes to make a quote request, `New Quote Request` event executes and creates a `QuoteRequest` in state `PendingPriceQR`. This `PendingPriceQR` has an attribute, `deadline`, which establishes the last day in which the customer is wishing to wait for a price. If this deadline is not met, then the `PendingPriceQR` is automatically rejected and changes its state to `RejectedQR`. Notice that `at(self.deadline)` is a time event, which results in the execution of effect `Autoreject QR`.

On the other hand, if the price for the request is established on time, it changes its state to `PendingConfirmationQR`, as now the quote request is waiting for the customer to decide whether he accepts the price or not. In both cases, event `Make Decision` executes, and depending on the outcome of this event, the quote request changes its state to `AcceptedQR` (condition `success`) or to `RejectedQR` (condition `failure`). Eventually, an `AcceptedQR` will be processed (event `Create Supplier Order`) and the requested CDs ordered to the supplier, prompting a change of state to `OrderedToSuppQR`.

An `OrderedToSuppQR` will change state to `ProcessedQR` when it is sent to the customer (event `Send Items`). Notice that this will only happen when the condition[2] is met: all the orders containing products in the quote request must have been received. Finally, the quote request is closed (state `ClosedQR`) after the customer receives the order, indicated by `Close QR` event.

3.3 Activity Diagrams

As we have explained previously, each external event in the state machine diagram would have the corresponding activity diagram showing its details. Bearing this in mind, for the state machine diagram of `Order`, we would have the following activity diagrams: `New Order`, `Send to Supplier`, `Receive Order`. For the state machine diagram of `QuoteRequest`, we would have the following activity diagrams: `New Quote Request`, `Calculate Price`, `Make Decision`, `Create Supplier Order`, `Send Items`, `Close QR`.

As there are many activity diagrams, we will focus on those that are more useful to illustrate the characteristics of our approach. In particular, we will look at the following diagrams: `Create Supplier Order`, `Send Items`, `Make Decision`. The rest of diagrams can be found in the appendix at the end.

Create Supplier Order. Figure 5 depicts the activity diagram of `Create Supplier Order`. It first starts the order creation process, and afterwards it manages the assignment of the items in an `AcceptedQR` to the right `Order`. As each CD is provided by one supplier, the activity diagram checks if there is an `OpenOrder` for the given supplier. If there is not, it calls activity diagram `New Order`. In any case, it obtains the `OpenOrder` and links it to the current `QuoteRequest`. When there are no CDs left to process, the activity diagram ends.

Notice that the node in charge of creating the new order is in fact a *subprocess* and it is decomposed in another activity diagram, as indicated by the rake-like symbol on the right-hand side of the node. In fact, this activity diagram corresponds to event `New Order` in the state machine diagram of `Order`. In this particular example, this is how the evolution of the two artifacts is related: when linking the quote request to a supplier order, if there is no available order for the required supplier, a new order is created.

Send Items. Figure 6 shows the activity diagram for event `Send Items` in the state machine diagram of `Quote Request`. It represents the process of sending the CDs, once they have been received from the supplier(s), to the customer.

[2] Condition "All orders received" is defined in OCL as: `self.order -> forAll(o | o.oclIsTypeOf(ReceivedOrder))`

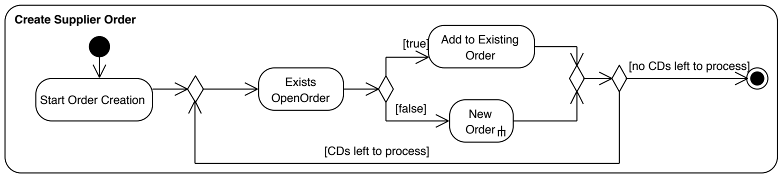


Fig. 5. Activity diagram of `Create Supplier Order`.

First of all, the necessary items for the quote request are picked up from the warehouse (`Obtain Items from Warehouse`). After this, they are packed up and sent to the customer (`Pack Items` and `Send Package`). Once they have been physically sent, the quote request is marked as sent. Notice that this event is made up of three material actions and one task. The three actions represent particular physical tasks that are carried out in the process but that do not directly make changes to the system. The only task that makes changes to the system is the last one, `Mark as Sent`.

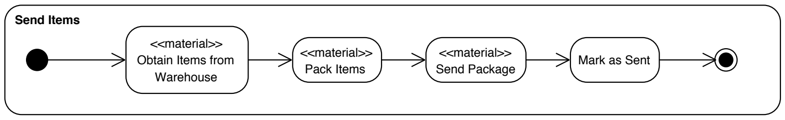

Send Items
<<material>> Obtain Items from Warehouse
<<material>> Pack Items
<<material>> Send Package
Mark as Sent

Fig. 6. Activity diagram of `Send Items`.

Make Decision. We include a final diagram in Figure 7 to illustrate the use of stereotypes in the activity diagram and how they connect to the state machine diagram. The activity diagram corresponds to event `Make Decision` of `QuoteRequest`. It basically represents the user's decision to either accept the quote request or to reject it. Depending on the user's decision, either task `Accept QuoteRequest` or task `Reject QuoteRequest` executes. The activity diagram ends in stereotype `succeed` in the first case or `fail` in the second, which connect directly with the event-dependent conditions in the state machine diagram of `QuoteRequest`.

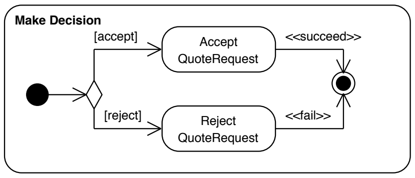


Fig. 7. Activity diagram of `Make Decision`.

3.4 Operation Contracts

This section presents the OCL operation contracts of some of the tasks in our example. In particular, it will focus on the tasks that belong to activity diagram `Make Decision` and on the specification of the only effect that we have in the state machine diagram of `QuoteRequest`: `Autoreject QR`.

Listing 1. Code for task *AcceptQuoteRequest*

```
action AcceptQuoteRequest(quoteID: Natural)
localPre: -
localPost:
let quote: QuoteRequest =
    QuoteRequest.allInstances()->select(qr | qr.id=quoteID) in
quote.oclIsTypeOf(AcceptedQR) and not
    (quote.oclIsTypeOf(PendingConfirmationQR)) and
    quote.oclAsType(AcceptedQR).date=today() and
    quote.oclAsType(AcceptedQR).finalPrice =
    quote@pre.oclAsType(PendingConfirmationQR).finalPrice
```

Listing 1 shows the operation contract of service `Accept QuoteRequest`. It has as input parameter the `quoteID` of the `QuoteRequest` that the customer wishes to accept. Then the service changes the state of the `QuoteRequest` to `AcceptedQR` and stores the final price and the date in which the `QuoteRequest` has been accepted.

Listing 2. Code for task *RejectQuoteRequest*

```
action RejectQuoteRequest(quoteID: Natural, reason:String)
localPre: -
localPost:
let quote: QuoteRequest =
    QuoteRequest.allInstances()->select(qr | qr.id=quoteID) in
quote.oclIsTypeOf(RejectedQR) and not
    quote.oclIsTypeOf(PendingConfirmationQR)) and
    quote.oclAsType(RejectedQR).date=today() and
    quote.oclAsType(RejectedQR).reason=reason
```

Listing 2 shows the OCL code for `Reject QuoteRequest`. Given a `quoteID` identifying a `QuoteRequest` and a reason for the rejection as input, it changes the `QuoteRequest` to state `RejectedQR`, storing the date in which the decision was made and the reason for the rejection (given as input).

Finally, we believe it is interesting to look at the specification of `Autoreject QR`. Remember that this effect executes when time event `at(self.deadline)` takes place, that is, when the deadline established by the customer is reached and a `PendingPriceQR` has not changed its state because the shop has not established a price.

Listing 3 shows the OCL code for the effect. It has as input the quote request which has reached the deadline, and the postcondition changes its state to `RejectedQR` stating the reason for the change.

Listing 3. Code for task *AutorejectQR*

```
action RejectQuoteRequest(quote: QuoteRequest)
localPre: -
localPost:
quote.oclIsTypeOf(RejectedQR) and not
    (quote.oclIsTypeOf(PendingPriceQR)) and
    quote.oclAsType(RejectedQR).date=today() and
    quote.oclAsType(RejectedQR).reason=‘‘Deadline reached’’
```

4 Related Work

In this section we analyze different alternatives to represent business process models. We first begin by examining process-centric approaches, and afterwards we look at artifact-centric alternatives.

4.1 Process-centric Approaches

There are several languages available to represent business process models following a traditional, or process-centric, approach. One of the most well-known is probably BPMN (Business Process Modeling Notation); however, there are several others such as UML activity diagrams, Workflow nets or YAWL (Yet Another Workflow Language) [20]. Although some of these languages have the ability to represent the data needed in the flow, their focus is on the sequencing of the tasks that are carried out in the process. DFDs (data-flow diagrams) would be one example of this. Although they place high importance on the data, the focus is on how these data move in the process, from one task to next, and little importance is given to their details or on the precise meaning of the tasks [21].

Another well-known language is BPEL (Business Process Execution Language). However, it is meant to be a web-service composition language following XML notation, and our focus is on defining processes at a high level of abstraction.

There are some process-centric works that do take data into consideration. For instance, [22] represents the associations between services in a WFD-net (WorkFlow nets annotated with Data). The tasks are annotated with the data that is created, read or written by the task. Similarly, [23] uses WSM nets which represent both the control flow and the data flow, although the data flow is limited to read and write dependencies between activities and data. [24] represents associations in an operational model, which shows tasks (or services) as nodes connected using arrows or edges. The operational model also shows the transfer of artifacts between tasks by indicating them over the edges. However, details of artifacts are not shown.

4.2 Artifact-centric Approaches

After giving an overview of process-centric approaches, we will deal with works that specify business processes from an artifact-centric perspective.To facilitate the analysis, this subsection is structured according to the dimensions of the BALSA framework for easier readability and comparison. At the end of the subsection we include a table which summarizes our analysis.

Business Artifacts Business artifacts can be represented in several ways. Many authors opt for a database schema [25, 11, 26, 27, 7], while others consider artifacts as a set of attributes or variables [28, 12, 29, 3]. Another alternative is to add an ontology represented by means of description logics on top of a relational database [30]. Although some of these alternatives describe the artifacts in a formal way, none of them represent the artifacts in a graphical way. This has some disadvantages: the models are more difficult to understand, e.g. it is more difficult to see how the artifacts relate to one another and to other objects.

There are also many works that represent artifacts in a graphical and formal way. For instance, [4, 5, 6] represent the business artifact and its lifecycle in one model, GSM, that includes the artifact's attributes. However, the relationships between artifacts are not made explicit. On the other hand, [31] represents artifacts as state machine diagrams defined by Petri nets, but does not give details on how the attributes of an artifact are represented. Closer to a UML class diagram is the Entity-Relationship model used in [32]. [16] uses a UML class diagram. Both the ER diagram and the UML class diagram are graphical and formal (or semi-formal) alternatives.

Finally, [8] defines its own framework, the PHILharmonicFlows, which uses a diagram that falls in-between a UML diagram and a database schema representation. Although it is a semi formal representation, it has the drawback of not using any well-known languages.

Lifecycles The lifecycle of a business artifact may be implicitly represented by using dynamic constraints in logic [25] or the tasks (or actions in the terminology of the papers) that make changes to the artifacts [26, 27, 11, 30]. [7] derives the artifact's lifecycle from a BPMN model annotated with data.

In this context, however, we are interested in approaches that represent the lifecycles explicitly. In many cases, such as [32], they are based on state machine diagrams, as they show very clearly the states in the evolution of the artifact and how each state is reached and under which conditions.

The GSM approach is a similar alternative to state machine diagrams, as it also represents in a graphical way the stages in the evolution of an artifact and the guard conditions, but adding the concept of milestone to them. A milestone is a condition that, once it is fulfilled, it closes a state. Another difference with state machine diagrams is that the sequencing of stages is determined by the guard conditions and not by edges connecting the states, making it much less straightforward than state machine diagrams. However, it is possible to use edges as a *macro*. GSM was first defined in [4] and further studied and formalized in [5, 6].

Another alternative to represent lifecycles is to use variants of Petri nets [16, 31, 33]. These representations are both graphical and formal. [8], within the PHILharmonicsFlows framework, uses a micro process to represent the evolution of an artifact and its states, which results in a graphical representation similar to GSM, without its strong formality.

Finally, some works opt for using a variable to store the artifact's state [12, 28]. Although it is an explicit representation, it only stores the current state of the artifact, instead of showing how it will evolve from one stage to the next. Therefore, it is a poorer form of representation in contrast to state machine diagrams, variants of Petri nets or GSM.

Associations In general, the different ways of representing associations can be classified on whether they represent them graphically or not. Many non-graphical alternatives are based on variants on condition-action rules. These alternatives have one main disadvantage over graphical ones: in order to know the order in which the tasks can execute, it will be necessary to carefully examine the rules. In contrast, graphical alternatives are easier to understand at a glance.

For instance, [25, 11, 27, 30] use a set of condition-action rules defined in logic. In [12], preconditions determine the execution of the actions; as such, they act as associations. As they are defined in logic, they are formal and unambiguous.

Likewise, [32] uses event-condition-action rules, but they are defined in natural language. Using natural language makes them easier to understand than those defined in logic, but they have a severe drawback: they are not formal and because of this they may have ambiguities and errors.

Alternatively, [16] uses *channels* to define the connections between *proclets*. A proclet is a labeled Petri net with ports that describes the internal lifecycle of an artifact. On the other hand, DecSerFlow allows specifying restrictions on the sequencing of tasks, and it is used in [33]. It is a language grounded on temporal logic but also includes a graphical representation.

When it comes to graphical representations, [8] uses micro and macro processes to represent the associations between the services. [7] uses a BPMN diagram to represent the associations between the tasks. In this sense, it is very similar to our proposal to use UML activity diagrams. All these approaches are graphical and formal.

In contrast, [3, 2] opt for a graphical representation using flowcharts and, because of this, the resulting models can be easily understood. However, they do not use any particular language to define the flow and they do not define the semantics of flowchart.

Services Services are also referred to as tasks or actions in the literature. In general, they are described by using pre and postconditions (also called effects). Different variants of logic are used in [25, 11, 12, 28, 29, 26, 30] for this purpose. [27, 5] omit the preconditions. The use of logic implies that the definition of services is precise, formal and unambiguous, but it is hardly understandable by the people involved in the business process.

Conversely, [32] uses natural language to specify pre and postconditions. In contrast to logic, natural language is easy to understand, but it is an informal description of services: this implies that the service definition may be ambiguous and error-prone.

Finally, [7] expresses the preconditions and postconditions of services by means of data objects associated to the services. These data objects are annotated with additional information such as what is read or written. [8] defines "micro steps" in the stages of their model which correspond to attributes that are modified. Neither of this two proposals are as powerful as using logic nor OCL operation contracts.

Table 1. Overview of alternative representations of data-centric process models. P.F stands for *PHILharmonicFlows*

		Approach	**Graphical?**	**Formal?**
Artifacts	DB Schemas	[25, 27, 26, 11, 7]		✓
	Attributes	[12, 28, 29, 3]		
	Ontology	[30]		✓
	ER Model	[32]	✓	✓
	UML Class Diagram	[16]	✓	✓
	Data diagr. (P.F.)	[8]	✓	✓
	GSM's attributes	[4, 5, 6]		✓
	Petri-Nets	[31]	✓	✓
Lifecycle	State Machine	[32]	✓	✓
	Variants of Petri-Nets	[33, 16, 31]	✓	✓
	GSM	[4, 5, 6]	✓	✓
	Micro proc. (P.F.)	[8]	✓	✓
	Variable	[12, 28]		
Associations	CA Rules - Logic	[25, 11, 27, 30]		✓
	Preconditions	[12]		✓
	ECA Rul. - Nat. L.	[32]		
	DecSerFlow	[33]	✓	✓
	Channels (Proclets)	[16]	✓	✓
	Micro/macro proc. (P.F.)	[8]	✓	✓
	BPMN	[7]	✓	✓
	Flowcharts	[3, 2]	✓	
Serv.	Pre / Post. in Logic	[25, 12, 28, 29, 26, 30, 11, 27, 5]		✓
	Natural Language	[32]		
	Micro steps (P.F.)	[8]	✓	✓
	Data Objects	[7]	✓	

Summary To conclude this section, Table 1 shows a summary of the artifact-centric approaches. As the table shows, none of the analyzed approaches uses the same language to represent all these dimensions in artifact-centric business processes. In many cases, the chosen system of representation is not graphical, which makes the models more difficult to understand. To complicate matters further, in many instances the language that is used is grounded on logic. Although formal, it is not understandable by business people. Natural language, on the other hand, is not a good option either: it can be easily understood, but it may lead to ambiguities and errors.

5 Conclusions

In this paper we have presented a methodology to model business process models from an artifact-centric perspective. To do so we have used the BALSA framework as a basis, proposing a different model for each dimension in the framework. As we have seen, the artifact-centric approach to business process modeling considers the data needed by the process, and because of this, it is possible to define formally the meaning of the tasks in the process.

To represent the diagrams in the example we have opted for a combination of models using the UML and OCL languages, because they integrate naturally and they give an homogeneous view (as it uses the same language) for the business. These languages can be understood by domain experts and they provide a high level of abstraction. Another advantage of using these combination of models is that, as we have shown in previous work [9], it is possible to perform semantic reasoning on the models to ensure that they fulfill the user requirements.

However, as long as the semantics of our models are respected, other alternatives are viable, with the same results, as we have outlined in this paper. Moreover, it is also possible to establish restrictions over these models to ensure that the verification that can be performed on them is decidable [10].

We have illustrated our approach by means of an example based on a CD online shop. This complexity in this example lies in the fact that there is a many-to-many relationship between the two artifacts in the model.

As further work, we would like to create a tool that given these set of models, is able to automatically check their correctness. In addition, it would also be interesting to carry out user-defined tests for those requirements that cannot be directly inferred from the model.

Acknowledgements

This work has been partially supported by the Ministerio de Ciencia e Innovación under project TIN2011-24747 and by Universitat Politècnica de Catalunya.

References

1. Hull, R.: Artifact-centric business process models: Brief survey of research results and challenges. In Meersman, R., Tari, Z., eds.: OTM 2008. Volume 5332 of LNCS. Springer Berlin / Heidelberg (2008) 1152–1163

2. Bhattacharya, K., Caswell, N.S., Kumaran, S., Nigam, A., Wu, F.Y.: Artifact-centered operational modeling: lessons from customer engagements. IBM Syst. J. **46**(4) (October 2007) 703–721
3. Nigam, A., Caswell, N.S.: Business artifacts: an approach to operational specification. IBM Syst. J. **42**(3) (2003) 428–445
4. Hull, R., et al.: Introducing the Guard-Stage-Milestone Approach for Specifying Business Entity Lifecycles. In Bravetti, M., Bultan, T., eds.: WS-FM 2010. Volume 6551 of LNCS. (2011) 1–24
5. Damaggio, E., Hull, R., Vaculín, R.: On the equivalence of incremental and fixpoint semantics for business artifacts with Guard – Stage – Milestone lifecycles. Information Systems **38**(4) (2013) 561 – 584 Special section on BPM 2011 conference.
6. Hull, R., et al.: Business artifacts with guard-stage-milestone lifecycles: managing artifact interactions with conditions and events. In Eyers, D.M., Etzion, O., Gal, A., Zdonik, S.B., Vincent, P., eds.: DEBS, ACM (2011) 51–62
7. Meyer, A., Pufahl, L., Fahland, D., Weske, M.: Modeling and enacting complex data dependencies in business processes. In Daniel, F., Wang, J., Weber, B., eds.: Business Process Management - 11th International Conference, BPM 2013, Beijing, China, August 26-30, 2013. Proceedings. Volume 8094 of Lecture Notes in Computer Science., Springer (2013) 171–186
8. Künzle, V., Reichert, M.: Philharmonicflows: towards a framework for object-aware process management. Journal of Software Maintenance **23**(4) (2011) 205–244
9. Estañol, M., Sancho, M., Teniente, E.: Reasoning on UML data-centric business process models. In Basu, S., Pautasso, C., Zhang, L., Fu, X., eds.: Service-Oriented Computing - 11th International Conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings. Volume 8274 of Lecture Notes in Computer Science., Springer (2013) 437–445
10. Calvanese, D., Montali, M., Estañol, M., Teniente, E.: Verifiable UML artifact-centric business process models. In Li, J., Wang, X.S., Garofalakis, M.N., Soboroff, I., Suel, T., Wang, M., eds.: CIKM 2014, ACM (2014) 1289–1298
11. Bagheri Hariri, B., et al.: Verification of relational data-centric dynamic systems with external services. In: PODS, ACM (2013) 163–174
12. Damaggio, E., Deutsch, A., Vianu, V.: Artifact systems with data dependencies and arithmetic. ACM Trans. Database Syst. **37**(3) (2012) 22
13. Gerede, C.E., Su, J.: Specification and verification of artifact behaviors in business process models. In Krämer, B.J., Lin, K.J., Narasimhan, P., eds.: ICSOC 2007. Volume 4749 of LNCS., Springer (2007) 181–192
14. Estañol, M., Queralt, A., Sancho, M.R., Teniente, E.: Artifact-centric business process models in UML. In La Rosa, M., Soffer, P., eds.: Business Process Management Workshops 2012. Volume 132 of LNBIP., Springer (2013) 292–303
15. Estañol, M., Queralt, A., Sancho, M.R., Teniente, E.: Using UML to specify artifact-centric business process models. In: BMSD 2014 : Proceedings of the Fourth International Symposium on Business Modeling and Software Design, SciTePress (2014) 84–93
16. Fahland, D., Leoni, M.D., van Dongen, B.F., van der Aalst, W.M.P.: Behavioral conformance of artifact-centric process models. In Abramowicz, W., ed.: BIS 2011. Volume 87 of LNBIP., Springer (2011) 37–49
17. Olivé, A.: Conceptual Modeling of Information Systems. Springer, Berlin (2007)
18. ISO: ISO/IEC 19505-2:2012 - OMG UML superstructure 2.4.1 (2012) Available at: `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=52854`.

19. Queralt, A., Teniente, E.: Specifying the semantics of operation contracts in conceptual modeling. In: Journal on Data Semantics VII. Volume 4244 of LNCS. Springer Berlin / Heidelberg (2006) 33–56
20. Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer, Berlin Heidelberg (2007)
21. Yourdon, E.: Just enough structured analysis (2006) Available at: http://www.yourdon.com/jesa/pdf/JESA_p.pdf.
22. Trcka, N., Aalst, W.M.P.V.D., Sidorova, N.: Data-Flow Anti-patterns : Discovering Data-Flow Errors in Workflows. In van Eck, P., Gordijn, J., Wieringa, R., eds.: CAiSE 2009. Volume 5565 of LNCS. (2009) 425–439
23. Ly, L.T., Rinderle, S., Dadam, P.: Semantic Correctness in Adaptive Process Management Systems. In Dustdar, S., Fiadeiro, J., Sheth, A., eds.: BPM 2006. Volume 4102 of LNCS., LNCS (2006) 193–208
24. Liu, R., Bhattacharya, K., Wu, F.Y.: Modeling Business Contexture and Behavior Using Business Artifacts. In Krogstie, J., Opdahl, A., Sindre, G., eds.: CAiSE 2007. Volume 4495 of LNCS., Springer (2007) 324–339
25. Bagheri Hariri, B., Calvanese, D., De Giacomo, G., De Masellis, R., Felli, P.: Foundations of relational artifacts verification. In Rinderle-Ma, S., Toumani, F., Wolf, K., eds.: BPM 2011. Volume 6896 of LNCS., Springer (2011) 379–395
26. Belardinelli, F., Lomuscio, A., Patrizi, F.: Verification of deployed artifact systems via data abstraction. In Kappel, G., Maamar, Z., Nezhad, H.R.M., eds.: ICSOC 2011. Volume 7084 of LNCS., Springer Berlin Heidelberg (2011) 142–156
27. Cangialosi, P., Giacomo, G.D., Masellis, R.D., Rosati, R.: Conjunctive artifact-centric services. In Maglio, P.P., Weske, M., Yang, J., Fantinato, M., eds.: ICSOC 2010. Volume 6470 of LNCS., Springer (2010) 318–333
28. Bhattacharya, K., Gerede, C., Hull, R., Liu, R., Su, J.: Towards formal analysis of artifact-centric business process models. In Alonso, G., Dadam, P., Rosemann, M., eds.: BPM 2007. Volume 4714 of LNCS., Springer (2007) 288–304
29. Fritz, C., Hull, R., Su, J.: Automatic construction of simple artifact-based business processes. In Fagin, R., ed.: ICDT 2009. Volume 361., ACM (2009) 225–238
30. Calvanese, D., Giacomo, G.D., Lembo, D., Montali, M., Santoso, A.: Ontology-based governance of data-aware processes. In Krötzsch, M., Straccia, U., eds.: RR. Volume 7497 of LNCS., Springer (2012) 25–41
31. Lohmann, N., Wolf, K.: Artifact-Centric Choreographies. In Maglio, P.P., Weske, M., Yang, J., Fantinato, M., eds.: ICSOC 2010. Volume 6470 of LNCS., Springer (2010) 32–46
32. Bhattacharya, K., Hull, R., Su, J.: A Data-Centric Design Methodology for Business Processes. In: Handbook of Research on Business Process Management. (2009) 1–28
33. Kucukoguz, E., Su, J.: On lifecycle constraints of artifact-centric workflows. In Bravetti, M., Bultan, T., eds.: WS-FM 2010. Volume 6551 of LNCS., Springer (2011) 71–85

Appendices

A Specification of the Whole Example

This appendix presents the whole specification of the example following our approach. For easier reference, we include again the class diagram, state machine diagrams and activity diagrams (with the respective operation contracts) for those elements already described in the paper.

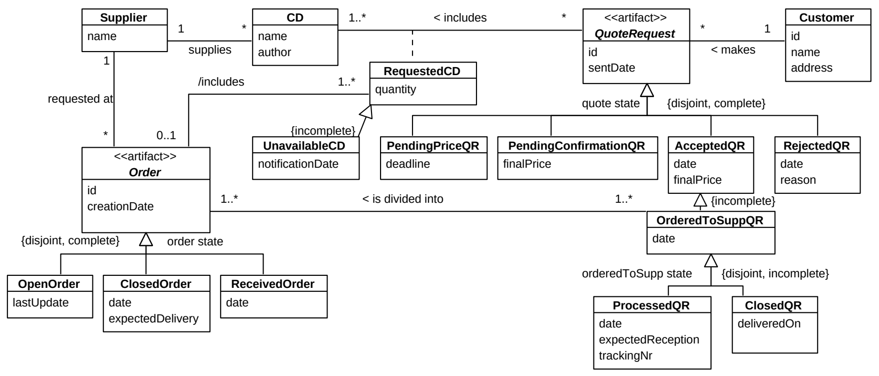
1 Class Diagram

Figure 8 shows the class diagram for our example. There are two business artifacts: `QuoteRequest` and `Order`, as shown by the stereotypes. The rest of the classes in the diagram, such as `Supplier`, `Customer` or `CD` represent objects: relevant information for the business but whose evolution we do not track. Each artifact has its own identifier, in this case, for both `QuoteRequest` and `Order` the identifier is `id`. The rest of classes in the diagram may also have their identifiers, for instance, a `CD` is identified by both its `name` and `author`. Each artifact and object has as many attributes and relationships as relevant for the business.

Artifact `Order` is the simpler of the two. It has three different subclasses: `OpenOrder`, `ClosedOrder` and `ReceivedOrder` which contain the relevant information for that particular state of the artifact. An `OpenOrder` is waiting to be sent to the supplier and additional `QuoteRequest`s can be assigned to it. A `ClosedOrder` has already been sent to the supplier. Finally, an `Order` changes its state to `ReceivedOrder` when it has been received at the shop.

On the other hand, artifact `QuoteRequest` has a first level of subclasses which are `PendingPriceQR`, `PendingConfirmationQR`, `AcceptedQR` and `RejectedQR`. A `PendingPriceQR` is waiting for the shop to quote the price. A `PendingConfirmationQR` has already a price and is waiting for the customer's acceptance or rejection. An `AcceptedQR` has already been accepted by the customer. In contrast, `RejectedQR` has been rejected.

`AcceptedQR` has one subclass: `OrderedToSuppQR`. Notice that the hierarchy is incomplete. An `OrderedToSuppQR` has already been split into several `Order`s that will eventually be processed and sent to the suppliers. At the same time, `OrderedToSuppQR` has two subclasses: `ProcessedQR` and `ClosedQR`, and the hierarchy is incomplete. Like in the previous case, an `OrderedToSuppQR` may not have any of the subtypes. `ProcessedQR` represents a quote request that has already been sent to the customer, and a `ClosedQR` corresponds to a quote request that has already been received by him or her.



1. Identifiers: (QuoteRequest, id), (Customer, id), (CD, name+author), (Supplier, name), (Order, id)
2. A ProcessedQR may not have more than one Order for the same Supplier.
3. Derived Relationship includes: A RequestedCD will be related to an Order if the QuoteRequest of the RequestedCD is linked to an Order requested at its CD Supplier.
4. There can only be at most one OpenOrder per Supplier.
5. The Orders into which a ProcessedQR is divided must be to Suppliers who provide the CDs included in the ProcessedQR.
6. The *sentDate* of a *QuoteRequest* must be earlier than attribute *date* in any of the subclasses of *QuoteRequest*.
7. *QuoteRequest.sentDate* < *PendingPriceQR.deadline*
8. *AcceptedQR.date* ≤ *OrderedToSuppQR.date*
9. *OrderedToSuppQR.date* ≤ *ProcessedQR.date*
10. *ProcessedQR.date* ≤ *ClosedQR.deliveredOn*
11. The *creation* date of *Order* must be earlier or equal to attribute *date* in any of its subclasses.
12. *Order.creationDate* ≤ *OpenOrder.lastUpdate*

Fig. 8. Class diagram showing the business artifacts as classes with the corresponding integrity constraints

2 State Machine Diagrams

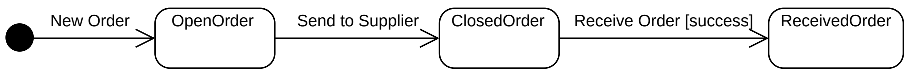


Fig. 9. State machine diagram for artifact `Order`.

Figures 9 and 10 show the state machine diagrams that correspond to the business artifacts in this example: `Order` and `QuoteRequest`. We will begin by looking at the state machine diagram for `Order`, which is simpler. In this case, there is a single-level hierarchy in the class diagram with restrictions `disjoint` and `complete`, therefore the states exactly map to the subclasses in the class diagram. An `Order` is created when there is a request to create a new order, as shown by event `New Order`. This order remains in state `OpenOrder`

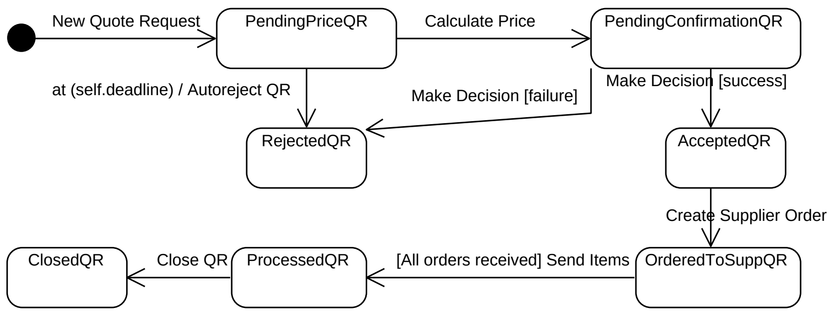


Fig. 10. State machine diagram for artifact `QuoteRequest`.

until someone decides that the order can be made to supplier, by executing event `Send to Supplier`. Then the order becomes a `ClosedOrder` and no more `AcceptedQR`s can be linked to it. Finally, once the order is received, if event `Receive Order` executes successfully, as indicated by tag `success`, it changes its state to `ReceivedOrder`.

On the other hand, artifact `QuoteRequest` has a more complex state machine diagram. First of all, it has a multi-level hierarchy. More specifically, it has three different levels. The first level has constraints `disjoint` and `complete`, but the second and third levels are `incomplete`. In the first-level hierarchy, the states are: `PendingPriceQR`, `PendingConfirmationQR`, `RejectedQR` and `AcceptedQR`. Although `AcceptedQR` has two subclasses, it is included because the hierarchy is incomplete, and therefore, there can exist an `AcceptedQR` which has no subtypes.

When a customer wishes to make a quote request, `New Quote Request` event executes and creates a `QuoteRequest` in state `PendingPriceQR`. This `PendingPriceQR` has an attribute, `deadline`, which establishes the last day in which the customer is wishing to wait for a price. If this deadline is not met, then the `PendingPriceQR` is automatically rejected and changes its state to `RejectedQR`. Notice that `at(self.deadline)` is a time event, which results in the execution of effect `Autoreject QR`.

On the other hand, if the price for the request is established on time, it changes its state to `PendingConfirmationQR`, as now the quote request is waiting for the customer to decide whether he accepts the price or not. In both cases, event `Make Decision` executes, and depending on the outcome of this event, the quote request changes its state to `AcceptedQR` (condition `success`) or to `RejectedQR` (condition `fail`). Eventually, an `AcceptedQR` will be processed (event `Create Supplier Order`) and the requested CDs ordered to the supplier, prompting a change of state to `OrderedToSuppQR`.

An `OrderedToSuppQR` will change state to `ProcessedQR` when it is sent to the customer (event `Send Items`). Notice that this will only happen when the condition[3] is met: all the orders containing products in the quote request must

[3] Condition "All orders received" is defined in OCL as: `self.order -> forAll(o | o.oclIsTypeOf(ReceivedOrder))`

have been received. Finally, the quote request is closed (state `ClosedQR`) after the customer receives the order, indicated by `Close QR` event.

3 Activity Diagrams and Operation Contracts

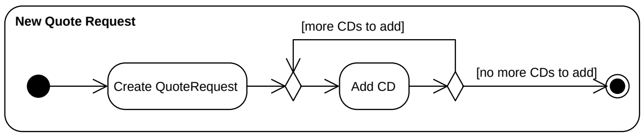


Fig. 11. Activity diagram of *New Quote Request*.

New Quote Request *New Quote Request* is in charge of creating a new quote request, and while there are CDs to add to the *QuoteRequest*, it keeps adding them.

Create QuoteRequest

```
action CreateQuoteRequest(cust: String, quoteID: Natural,
    deadlineDate: Date)
localPre: not(QuoteRequest.allInstances()->exists(qr |
    qr.id=quoteID))
localPost: PendingPriceQR.allInstances()->exists(qr |
    qr.oclIsNew() and qr.id=quoteID and qr.customer.id=cust and
    qr.sentDate=today() and qr.deadline=deadlineDate)
```

Task *Create QuoteRequest* creates a new *QuoteRequest* for the given customer and with the given `id` and `deadline`.

Add CD

```
action AddCD(cdName: String, cdAuthor: String, qty:Natural,
    quoteID: Natural)
localPre: not(RequestedCD.allInstances()->exists(rcd |
    rcd.cD.name=cdName and rcd.cD.author=cdAuthor and
    rcd.quoteRequest.id=quoteID))
localPost: RequestedCD.allInstances()->exists(rcd |
    rcd.oclIsNew() and rcd.quantity=qty and
    rcd.cD.author=cdAuthor and rcd.cD.name=cdName and
    rcd.quoteRequest.id=quoteID)
```

Add CD adds the given CD (identified by *cdName* and *cdAuthor*) in the given quantity to the order identified by *quoteID*.

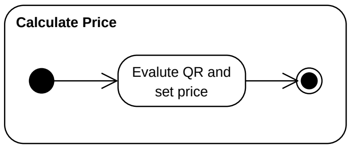


Fig. 12. Activity diagram of *Calculate Price.*

Calculate Price This activity diagram sets the price for the *QuoteRequest.*

Evaluate QR and Set Price

```
action EvaluateQRAndSetPrice(quoteID: Natural, price: Real)
localPre: -
localPost:
let quote: QuoteRequest =
    QuoteRequest.allInstances()->select(qr | qr.id=quoteID) in
quote.oclIsTypeOf(PendingConfirmationQR) and not
    quote.oclIsTypeOf(PendingPriceQR) and
    quote.oclAsType(PendingConfirmationQR).finalPrice=price
```

Task *EvaluateQRAndSetPrice* changes the state of the *QuoteRequest* to *PendingConfirmationQR* and sets its price to the given input *price.*

Make Decision Activity diagram *Make Decision* shows the decision making process after the price has been set for the *QuoteRequest.* If the customer wishes to accept the price, task *Accept* QuoteRequest executes. Otherwise, task *Reject* QuoteRequest executes.

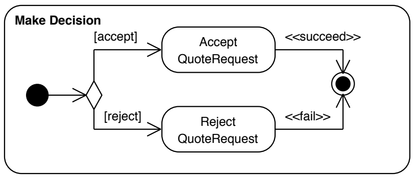


Fig. 13. Activity diagram of *Make Decision.*

Accept QuoteRequest

```
action AcceptQuoteRequest(quoteID: Natural)
localPre: -
localPost:
let quote: QuoteRequest =
    QuoteRequest.allInstances()->select(qr | qr.id=quoteID) in
```

```
quote.oclIsTypeOf(AcceptedQR) and not
    (quote.oclIsTypeOf(PendingConfirmationQR)) and
    quote.oclAsType(AcceptedQR).date=today() and
    quote.oclAsType(AcceptedQR).finalPrice =
    quote@pre.oclAsType(PendingConfirmationQR).finalPrice
```

Task *Accept QuoteRequest* has as input parameter the *quoteID* of the *QuoteRequest* that the customer wishes to accept. Then the service changes the state of the *QuoteRequest* to *AcceptedQR* and stores the final price and the date in which the *QuoteRequest* has been accepted.

Reject QuoteRequest

```
action RejectQuoteRequest(quoteID: Natural, reason:String)
localPre: -
localPost:
let quote: QuoteRequest =
    QuoteRequest.allInstances()->select(qr | qr.id=quoteID) in
quote.oclIsTypeOf(RejectedQR) and not
    quote.oclIsTypeOf(PendingConfirmationQR)) and
    quote.oclAsType(RejectedQR).date=today() and
    quote.oclAsType(RejectedQR).reason=reason
```

Given a *quoteID* identifying a *QuoteRequest* and a reason for the rejection as input, task *RejectQuoteRequest* changes the *QuoteRequest* to state *RejectedQR*, storing the date in which the decision was made and the reason for the rejection (given as input).

Create Supplier Order Figure 14 depicts the activity diagram of *Create Supplier Order*. It begins by starting the order creation process process. After this, for each CD in the quote request, it checks if there is an open order to the supplier of the CD in question. If there is, it adds the CD to the order. Otherwise, it creates a new order and adds the CD to it. When there are no CDs left to process, the activity diagram ends.

Notice that the node in charge of creating the new order is in fact a *subprocess* and it is decomposed in another activity diagram, as indicated by the rake-like symbol on the right-hand side of the node. In fact, this activity diagram corresponds to event *New Order* in the state machine diagram of *Order*. In this particular example, this is how the evolution of the two artifacts is related: when linking the quote request to a supplier order, if there is no available order for the required supplier, a new order is created.

Create Supplier Order manages the assignment of the items in an *AcceptedQR* to the right *Order*. As each CD is provided by one supplier, the activity diagram checks if there is an *OpenOrder* for the given supplier. If there is not, it calls activity diagram *New Order*. In any case, it obtains the *OpenOrder* and links it to the current *QuoteRequest*.

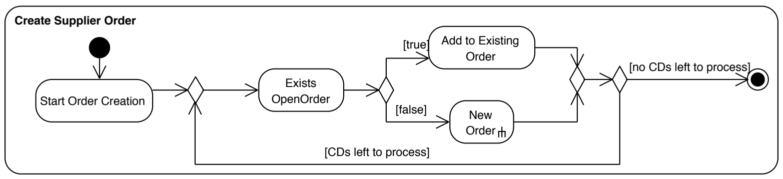


Fig. 14. Activity diagram of *Create Supplier Order*.

Start Order Creation

```
action StartOrderCreation (quoteID: Natural)
localPre: -
localPost:
let qr: QuoteRequest = QuoteRequest.allInstances()->select(qr |
    qr.id=quoteID) in
         qr.oclIsTypeOf(OrderedToSuppQR) and
             qr.oclAsType(OrderedToSuppQR).date=today()
```

Task *Start Order Creation* changes the state of the *QuoteRequest* to begin the process of relating the *OrderedToSuppQR* and the *Orders*.

Exists OpenOrder

```
action ExistsOpenOrder(quoteID: Natural, suppName: String):
    Boolean
localPre:
let quoteR = OrderedToSuppQR.allInstances()->select(qr |
    qr.id=quoteID) in
quoteR.order.supplier.name->excludes(suppName) and
    quoteR.cd.supplier.name->includes(suppName)
localPost: result = OpenOrder.allInstances()->exists(oo |
    oo.supplier.name=suppName)
```

Exists OpenOrder checks if there is an *OpenOrder* for the given supplier's name and returns a boolean value which will be `true` if there is indeed an *OpenOrder*. Notice that there is only one supplier per CD. Otherwise, it will return `false`. However, the task has a strong precondition. First of all, it ensures that the link between the given *QuoteRequest* and *Order* has not been created yet and that the given supplier actually supplies at least one of the CDs in the *QuoteRequest*.

New Order This is a call to another activity diagram. In this particular case, it triggers the transition to create a new order, in the state machine diagram of *Order*. The results is a new artifact *Order*.

Add to Existing Order

```
action AddToExistingOrder(quoteID: Natural, suppName: String)
localPre: -
localPost: OpenOrder.allInstances()->select(oo |
    oo.supplier.name).orderedToSuppQR.id->includes(quoteID)
```

AddToExistingOrder simply creates the link between the given *QuoteRequest*, identified by *quoteID*, and the given *Supplier*, identified by *suppName*.

Send Items *Send Items* represents the process of sending the CDs, once they have been received from the supplier(s), to the customer. The activity diagram has three material tasks: *Obtain Items from Warehouse*, *Pack Items* and *Send Package*. As their names imply, they represent the process of physically packing up the CDs and sending them to the customer. The last task is used to store in the system that the items have been sent and all the relevant information relating to that event.

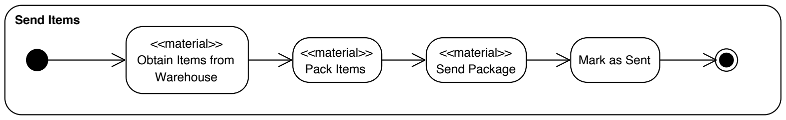

Send Items
<<material>> Obtain Items from Warehouse
<<material>> Pack Items
<<material>> Send Package
Mark as Sent

Fig. 15. Activity diagram of *Send Items*.

Mark as Sent

```
action MarkAsSent(quoteID: Natural, receptionDate: Date,
    trackNr: Natural)
localPre: -
localPost:
let quoteReq: QuoteRequest =
    QuoteRequest.allInstances()->select(qr | qr.id=quoteID) in
          quoteReq.oclIsTypeOf(ProcessedQR) and
              quoteReq.oclAsType(ProcessedQR).date=today() and
              quoteReq.oclAsType(ProcessedQR).expectedReception=receptionDate
              and
              quoteReq.oclAsType(ProcessedQR).trackingNr=trackNr
```

Task *Mark as Sent* changes the state of the *QuoteRequest* into *ProcessedQR* and stores the current date, the expected reception date and the tracking number of the package.

Close QR Activity diagram *Close QR* closes the *QuoteRequest* after it has been received by the customer. As shown on the diagram, it has only one task.

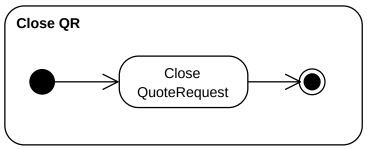


Fig. 16. Activity diagram of *Close QuoteRequest.*

Close QuoteRequest

```
action CloseQuoteRequest(quoteID: Natural, delivery: Date)
localPre: -
localPost:
let quoteReq: QuoteRequest =
    QuoteRequest.allInstances()->select(qr | qr.id=quoteID) in
          quoteReq.oclIsTypeOf(ClosedQR) and not
              quoteReq.oclIsTypeOf(ProcessedQR) and
              quoteReq.oclAsType(ClosedQR).deliveredOn=delivery
```

Close QuoteRequest changes the state of the *QuoteRequest* to *ClosedQR* and stores its delivery date.

3.1 Order

Order is the artifact that keeps information about the orders that are placed to the supplier. This subsection presents the activity diagrams showing the details of the events in its state machine diagram.

New Order Activity diagram *New Order* basically creates a new order.

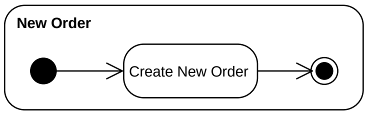


Fig. 17. Activity diagram of *New Quote Request.*

Create New Order

```
action CreateNewOrder(orderID: Natural, supplierName: String)
localPre: not(Order.allInstances()->exists(o | o.id=orderID))
localPost:
OpenOrder.allInstances()->exists(o | o.oclIsNew() and
    o.id=orderID and o.date=today() and
    o.supplier.name=supplierName and
    o.processedQR.id->includes(quoteID))
```

Create New Order creates a new order with the given id and for the supplier identified by *supplierName*. It also stores the date in which the order is created.

Send to Supplier This activity diagram closes the order and sends it to the provider. Afterwards, it registers the expected delivery date of the order.

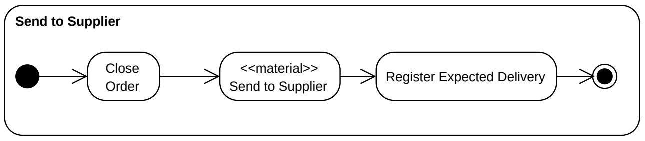


Fig. 18. Activity diagram of *Send to Supplier*.

Close Order

```
action CloseOrder(orderID: Natural, supplierName: String)
localPre: -
localPost:
let ord:Order = Order.allInstances()->select(o | o.id=orderID)
    in
ord.oclIsTypeOf(ClosedOrder) and not ord.oclIsTypeOf(OpenOrder)
    and ord.oclAsType(ClosedOrder).date=today()
```

Task *Close Order*, as the name implies, closes an *OpenOrder* and stores the date in which it is closed.

Register Expected Delivery

```
action RegisterExpectedDelivery(orderID: Natural, expectedDel:
    Date)
localPre:
localPost:
ClosedOrder.allInstances()->select(o |
    o.id=orderID).expectedDelivery=expectedDel
```

Register Expected Delivery stores the expected delivery date after the order has been sent.

Receive Order This activity diagram deals with the reception of an order.

Register Order as Received

```
action RegisterOrderAsReceived(orderID: Natural)
localPre: -
```

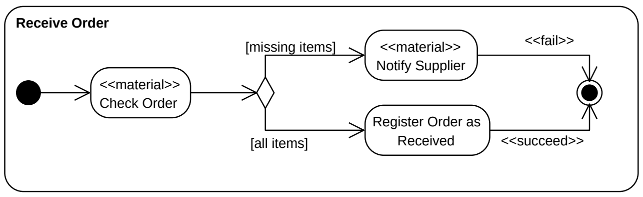


Fig. 19. Activity diagram of *ReceiveOrder*.

```
localPost:
let ord: Order = Order.allInstances()->select(o | o.id=orderID)
    in
ord.oclIsTypeOf(ReceivedOrder) and
    ord.oclAsType(ReceivedOrder).date=today()
```

Task *Register Order as Received* selects the order identified by *orderID* and changes its state to *ReceivedOrder* and stores the date of reception.