

Automatic Generation of Workflow-extended Domain Models (extended version)

Marco Brambilla¹, Jordi Cabot², Sara Comai¹

¹ Dipartimento di Elettronica e Informazione, Politecnico di Milano
Piazza L. Da Vinci, 32. I20133 Milano, Italy
{mbrambil, comai}@elet.polimi.it

² Estudis d'Informàtica, Multimèdia i Telecomunicació, Universitat Oberta de Catalunya
Rbla. del Poblenou, 156 E08018 Barcelona, Spain
jcabot@uoc.edu

Abstract. The specification of business processes is becoming a more and more critical aspect for organizations. Such processes are specified as workflow models expressing the logical precedence among the different business activities (i.e. the units of work). Up to now, workflow models have been commonly managed through specific subsystems, called workflow management systems. In this paper we advocate for the integration of the workflow specification in the system domain model. This workflow-extended domain model is automatically derived from the initial workflow specification. Then, model-driven development methods may depart from the extended domain model to automatically generate an implementation of the system enforcing the business processes in any final technology platform, thus avoiding the need of basing the implementation on a dedicated workflow engine.

1. Introduction

Software development processes for complex business applications usually require the definition of a workflow model to express logical precedence and process constraints among the different business activities (i.e. the units of work).

Currently, workflow models are usually implemented with the help of dedicated workflow management systems (e.g., [13], [19]) which are heavy-weight applications focused on the control aspects of the workflow enactment. Alternatively, some approaches focus on the implementation of the workflow model in a specific technology platform, as relational databases (generally in the form of triggers [2]), web applications (by means of hypertextual links and buttons properly placed in Web pages, thus restricting the user navigation [4]) or web services (through transformation into BPEL4WS [16]). These *ad hoc* approaches are hardly generalizable to other technologies.

In this paper we adopt a formalized model-driven development process for workflow-based applications and advocate for the automatic integration of the workflow model within the (platform-independent) domain model. Given a domain model d and a workflow model w , it is possible to automatically derive a full fledged domain

model d' enriched with the types needed to record the required workflow information in w (mainly its activities and the enactment of these activities in the different workflow executions) and with a set of process constraints over such types to control the correct workflow execution. We refer to this resulting model as the *workflow-extended domain model*. We will represent it using UML class diagrams annotated with OCL constraints to represent the process constraints. The whole process is sketched in Fig. 1.1. Note that, if necessary, several workflow models can be integrated within the same domain model. This approach has been implemented in a prototype tool.

The main characteristic of a workflow-extended domain model is that it automatically ensures a consistent behavior of all enterprise applications with respect to the business process specification. As long as the applications properly update the workflow information in the extended model, the generated process constraints enforce that the different tasks are done according to the initial workflow model.

Another advantage of a workflow-extended domain model is that it is platform-independent. Indeed, our workflow-extended model can benefit from any method or tool designed for managing a generic domain model, no matter the target technology platform or the purpose of the tool, spawning from direct application execution, to verification/validation analysis, to metrics measurement and to automatic code-generation in any final technology platform. Those methods do not need to be extended to cope with our workflow-extended models, since our workflow-extended domain model is a completely standard UML model.

Moreover, our workflow-extended models enable the definition of more expressive business constraints, including timing conditions [7] or involving both workflow and domain information. These constraints are generally not allowed by workflow definition languages.

The rest of the paper is structured as follows: in Section 2 the basic workflow concepts and our case study are illustrated. In Sections 3 and 4 we provide the definition of the workflow-extended domain model and of the OCL process constraints, respectively. Section 5 sketches possible implementation strategies for this extended model. Section 6 compares our approach with related work and in Section 7 we draw our conclusions, provide some details about our tool support and discuss future work.

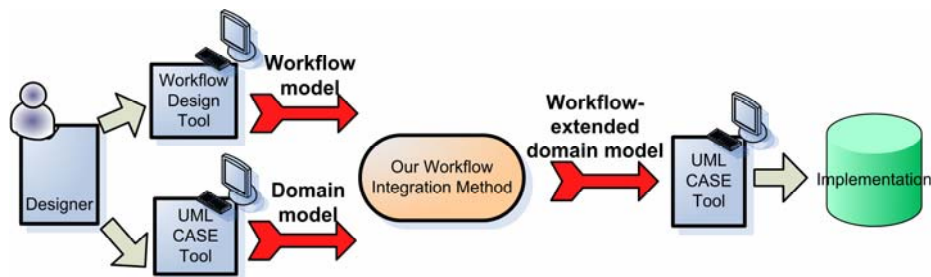


Fig. 1.1. MDD process for workflow-based applications

2. Basic workflow concepts

Several visual notations and languages have been proposed to specify workflow models, with different expressive power, syntax, and semantics. Without loss of generality, in our work we have adopted the *Workflow Management Coalition* terminology and the BPMN [18] OMG standard notation¹.

The workflow model is hence based on the concepts of *Process* (the description of the business process), *Case* (a process instance, that is, a particular workflow execution), *Activity* (the elementary unit of work composing a process), *Activity instance* (an instantiation of an activity within a case), *Actor* (a user role intervening in the process), *Event* (some punctual situation that happens in a case), and *Constraint* (logical precedence among activities and rules enabling activities execution). Processes can be internally structured using a variety of constructs: sequences of activities; gateways implementing AND, OR, XOR splits, respectively realizing splits into independent, alternative and exclusive threads; gateways implementing joins, i.e., convergence point of two or more activity flows; conditional flows between two activities; loops among activities or repetitions of single activities. Each construct may involve several constraints over the activities.

Our approach covers a large subset of the full expressive power of BPMN; we do not cope with the concepts of nested subprocesses (which can be easily tackled by flattening the process representation), transactions (which can exploit implementation features), and a few combinations of primitive constructs, such as the direct concatenation of several gateways (which can be handled by introducing fake activities between them).

In the sequel, we will exemplify the proposed approach on a case study consisting of a workflow implementing a simplified purchase process, as illustrated in Fig. 2.1.

According to the BPMN semantics, the depicted diagram specifies a process involving two actors (represented by the two swimlanes): a customer and a seller. The customer starts the workflow by asking for a quotation about a set of products (*Ask Quotation* activity). The seller provides the quotation (*Provide Quotation* activity) and the customer may decide (exclusive choice) to modify the request (and hence the quotation request and response are repeated) or to accept it (then the order is submitted and the seller takes care of it). For simplicity, it is not modeled what happens if they do never reach an agreement. The order management requires two parallel activities to be performed: the choice of the shipment options and the internal management of each order line. The latter is represented by the multi-instance activity called *Process OrderLine*: a different instance is started for each order line included in the order. Once all order lines have been processed and the shipment has been decided (i.e., after the AND merge synchronization), the order is shipped and the customer pays the corresponding amount.

¹ The results of our approach when using Activity Diagrams would have been quite similar. See [21] for a correspondence between BPMN and Activity Diagrams.

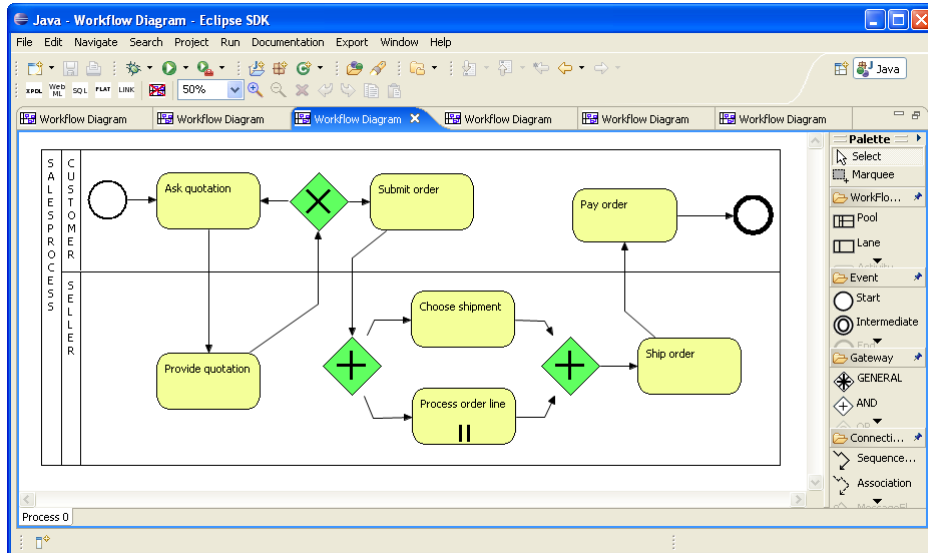


Fig. 2.1. Example of a workflow schema

3. Extending domain models with workflow information

Given an initial domain model, the workflow-extended domain model of the workflow-based application is obtained by extending the domain model with some additional elements derived from the workflow specification. We will focus on the case of a single workflow; however, our extensions to the domain model suffice when considering different workflows on the same domain.

Clearly, the workflow-extended domain model is more complex than the original domain model. However, we believe that this increased complexity is compensated by the fact that it may be automatically generated (with our method) and processed (with, for instance, code-generation tools) and thus, the designer does not need to manipulate it. Moreover, the size of the extension is constant regardless the size of the domain model and linear with respect to the number of activities in the workflow.

The workflow-extended model contains the minimum set of concepts required to manage the workflow and to easily specify the needed process constraints. However, richer schemas with further relationship types and/or attributes could be defined, according to the requirements of the specific workflow application (for example, we could have used a more complex pattern for the specification of the role-user relationship [5]). Similarly, simpler extensions could be used instead but then, as a trade-off, the process constraints would become much more complex.

To illustrate the process we will use the workflow model of Fig. 2.1 and we will assume that the initial domain model is the one shown in the bottom part of Fig. 3.1, consisting in the types *Product*, *Quotation*, *QuotationLine*, and *Order* (note that when

accepted by the customer, a *Quotation* generates an *Order* and then, its quotation lines are referred to as order lines).

The workflow-extended domain model must include at least: (i) the original domain model, (ii) *user*-related information, (iii) *workflow*-related information, (iv) a set of possible relationships between the domain schema, the workflow information and the user information, and (v) a set of process constraints guaranteeing a consistent state of the whole model with respect to the workflow definition (see the next section).

More formally, we define a workflow-extended domain model as follows. Given an initial domain model with entity types (i.e. classes) $E=\{e_1, \dots, e_n\}$, representing the knowledge about the domain, and a workflow model w with activities $A=\{a_1, \dots, a_m\}$, the workflow-extended domain model is obtained in the following way:

- i) *Domain subschema*: All entity types in E and their relationships (i.e. associations) remain unchanged in the workflow-extended model (bottom part of Fig. 3.1).
 - ii) *User subschema*: User-related information is added to the extended model by means of two entity types (see the top-left part of Fig. 3.1): entity type *User* represents individual workflow actors; entity type *Role* represents groups of users, having access to the same set of tasks. A user may belong to different roles.
 - iii) *Workflow subschema*: Workflow-related information (top-right part of Fig. 3.1) includes several fixed types (i.e. independent of the particular workflow model):
 - Entity type *Process* represents the supported workflows. As an example, an instance of the *Process* type would be our *Purchase* workflow. Other instances would represent additional workflows over the same domain subschema.
 - Entity type *Case* denotes an instance of a process, which has a name, a start time, an end time, and a status, which can be: ready, active, cancelled, aborted, or completed [18]. Every execution of a process results in a new instance of this type. This new instance is related with the appropriate *process* instance.
 - Entity type *ActivityType* represents the different activities that compose a process. Activity types are assigned to roles, which are responsible of executing them. In our case study, *AskQuotation*, *ProvideQuotation*, etc. would be instances of *ActivityType*.
 - Entity type *Activity* denotes the occurrence of a particular activity within a *Case*, described by the start time, the end time, and the current status, which can be: ready, active, cancelled, aborted, or completed. Only one user can execute a particular activity instance, and this is recorded by the relationship type *Performs*. The *Precedes* relationship keeps track of the execution order between activities.
 - Entity type *EventType* represents the events that may affect the sequence or timing of activities of a process (e.g., temporal events, messages etc.). There are three different kinds of events (*eventKind* attribute): start, intermediate, and end. For start and intermediate events we may define the triggering mechanism (*eventTrigger*). For end events, we may define how they affect the case execution (*eventResult*).
 - Entity type *Event* denotes the occurrence of a particular type of event in the system.
- and a set of workflow-dependent subtypes:

- For each activity $a \in A$, a new subtype s_a is added to the entity type *Activity* (*ActivityType* is a *powertype* for this set of generalization relationships). The name of the subtype is the name of a (e.g., in Fig. 3.1 we introduced *ProcessOrderLine*, *AskQuotation*, *ShipOrder*, and so on). These subtypes record the information about the specific activities executed during a workflow case. For instance, the action of asking a quotation for the purchase X in a case C of a workflow W would be recorded in the system as an instance of the *AskQuotation* subtype related with the corresponding instance “ C ” in the *Case* type (in its turn related with the “ W ” instance in the *Process* type)
- iv) *Relationships between workflow subschema and domain subschema*: each subtype s_a is related with a (possibly empty) set of entity types $E_a \subseteq E$. These new relationship types are useful to record the objects modified/managed during the execution of a certain activity. Also, they are required to evaluate conditions appearing in some process constraints. In the case study (see Fig. 3.1), a set of relationship types are established: *Quotations* are associated to the activities *Ask Quotation* and *Provide Quotation*; *QuotationLines* are associated to the *ProcessOrderLine* activity; and *Orders* are associated to the activities *Submit Order*, *Choose Shipment*, *Process OrderLine*, *Ship Order*, and *Pay Order*. When necessary, these associations between the domain and the workflow subschemata may be automatically generated if the workflow specification includes auxiliary primitives for describing the data flow between activities and/or when the designer defines some pattern-matching among the names of the activities and of the entity types. Otherwise, they must be manually specified

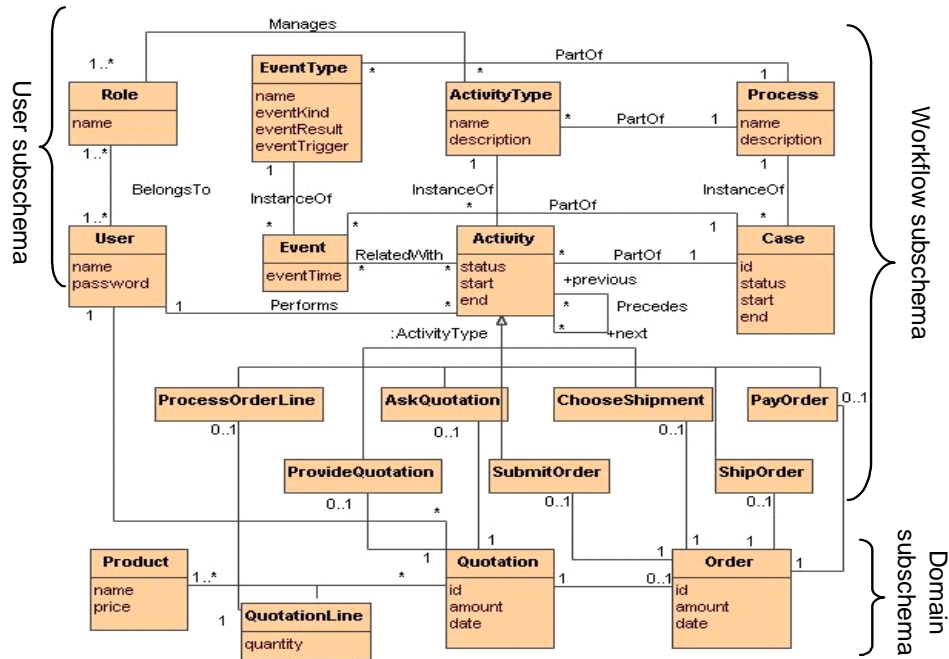


Fig. 3.1 - Workflow-extended domain model

4. Translation of process constraints

The structure of a workflow model implies a set of constraints regarding the execution order of the different activities, the number of possible instances of each activity in a given case, the conditions that must be satisfied in order to start a new activity, and so forth. These constraints are usually referred to as *process constraints*. The behavior of all enterprise applications must always satisfy these constraints. Thus, the generation of the workflow-extended model must consider all process constraints.

Process constraints are translated as constraints over the population of the s_{a1}, \dots, s_{am} subtypes of *Activity* (see previous section). The generated constraints guarantee that any update event over the population of one of these subtypes (for instance, the creation of a new activity instance or the modification of its status) will be consistent with the process constraints defined in the workflow model.

We specify process constraints by means of invariants written in the OCL language. Invariants in OCL are defined in the context of a specific type, the *context type*. The actual OCL expression stating the constraint condition is called the *body* of the constraint. The *body* is always a boolean expression and must be satisfied by all instances of the context type, that is, the evaluation of the body expression over every instance of the context type must return a *true* value. Constraints are defined to restrict only the execution of the workflow they are created for. Therefore, no interfer-

ences among different workflows occur, even if they are defined over an overlapping subset of the domain model.

The complexity of the constraints is of relative importance since all of them are automatically generated from the workflow model, and thus, they do not need to be manipulated (nor even necessarily understood) by the designer but for other tools. However, to simplify its presentation in the extended model, we could easily define an stereotype for each constraint type, as done in [8].

Next subsections define a set of patterns for the generation of the process constraints corresponding to the different constructs appearing in workflow models (sequences, split gateways, merge gateways, conditions, loops, and so on). The patterns can be combined to produce the full translation of the workflow model. As an example, we provide in Section 4.7 the translation of the workflow model of Fig. 2.1.

Note that some constructs admit several graphical representations equivalent to the ones used in this paper (see [18] for details). Moreover, the workflow language defines some complex constructs that can be derived from the basic ones, such as complex gateways and event-based gateways, not addressed here due to lack of space.

4.1 Sequences of activities

A sequence flow between two activities (Fig. 4.1) indicates that the first activity (*A*) must be completed before starting the second one (*B*). Moreover, if *A* is completed within a given case, *B* must be eventually started before ending the case (we do not require *B* to be completed since, for instance, it could be interrupted by the trigger of an intermediate exception event). This behavior can be enforced by means of the definition of three OCL constraints.

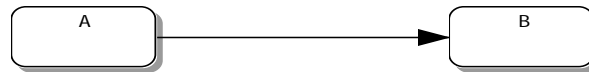


Fig. 4.1. Sequence flow

The first constraint (*seq₁* constraint) is defined over the entity type corresponding to the destination activity (*B* in the example) stating that for all activity instances of type *B* the preceding activity instance must be of type *A* and that it must have been already completed. Its specification in OCL is the following:

context B inv seq₁: previous->size()=1 and previous->exists(a| a.oclIsTypeOf(A) and a.status='completed')

This OCL definition enforces that *B* instances (since *B* is the context type of the constraint) have a previous activity (because of the *size* operator over the value of the navigation through the role *previous*) and that such activity is of type *A* (enforced by the *exists* operator). *B* and *A* are *Activity* subtypes as defined in Section 3.

The other two required constraints are:

- A constraint *seq₂* over the second activity to prevent the creation of two different *B* instances related with the same *A* activity instance

context B inv seq₂: B.allInstances()-> isUnique(previous)

- A constraint seq_3 over the *Case* entity type verifying that when the case is completed there exists a *B* activity instance for each completed *A* activity instance. This *B* instance must be the only instance immediately following the *A* activity instance.

context Case inv seq₃: status='completed' implies self.activity-> select(a| a.ocllsTypeOf(A) and a.status='completed')->forall(a/a.next->exists(b| b.ocllsTypeOf(B)) and a.next->size(=1)

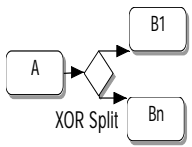
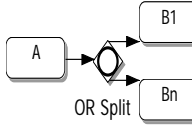
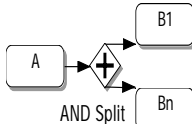
4.2 Split gateways

A split gateway is a location within a workflow where the sequence flow can take two or more alternative paths. The different split gateways differ on the number of possible paths that can be taken during the execution of the workflow. For *XOR-split* gateways only a single path can be selected. In *OR-splits* several of the outgoing flows may be chosen. For *AND-splits* all outgoing flows must be followed.

For each kind of BPMN split gateway, Table 4.1 shows the process constraints required to enforce the corresponding behavior.

Besides the process constraints appearing in the table, we must also add to all the activities $B_1...B_n$ the previous constraints seq_1 and seq_2 to verify that the preceding activity *A* has been completed and that no two activity instances of the same activity B_i are related with the same preceding activity *A*. We also require that the activity instance/s following *A* is of type B_1 or ... or B_n .

Table 4.1 Constraints for split gateways

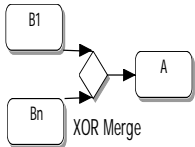
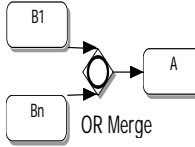
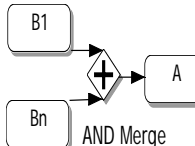
Split gateway	Process constraints
 <p>XOR Split</p>	<ul style="list-style-type: none"> - Only one of the $B_1..B_n$ activities may be started <p><i>context A inv: next->select(a a.ocllsTypeOf(B₁) or ... or a.ocllsTypeOf(B_n))->size(<=1</i></p> <ul style="list-style-type: none"> - If <i>A</i> is completed, at least one of the $B_1..B_n$ activities must be created before ending the case <p><i>context Case inv: status='completed' implies activities-> select(a a.ocllsTypeOf(A) and a.status='completed')-> forall (a a.next->exists(b b.ocllsTypeOf(B₁) or..or b.ocllsTypeOf(B_n)))</i></p>
 <p>OR Split</p>	<ul style="list-style-type: none"> - Since several $B_1..B_n$ activities may be started, we just need to verify that if <i>A</i> is completed, at least one of the $B_1..B_n$ activities is created before ending the case (like in the XOR split above)
 <p>AND Split</p>	<ul style="list-style-type: none"> -If <i>A</i> is completed all $B_1..B_n$ activities must be eventually started <p><i>context Case inv:status='completed' implies activity->select(a a.ocllsTypeOf(A) and a.status='completed')->forall(a a.next->exists(b b.ocllsTypeOf(B₁)) and ... and a.next->exists(b b.ocllsTypeOf(B_n)))</i></p>

4.3 Merge gateways

Merge gateways are useful to join or synchronize alternative sequence flows. Depending on the kind of merge gateway, the outgoing activity may start every time a single incoming flow is completed (*XOR-Merge*) or must wait until all incoming flows have finished in order to synchronize them (*AND-Merge* gateways). The semantics of the *OR-Merge* gateways is not so clear. If there is a matching *OR-split*, the *OR-Merge* should wait for the completion of all flows activated by the split. If no matching split exists several interpretations are possible, being the simplest one to wait just till the first incoming flow. This is the interpretation adopted in this paper. For a complete treatment of this construct see [22].

Table 4.2 presents the different translation patterns required for each kind of merge gateway. Besides the constraints included in the table, a constraint over *A* should be added to all gateways to verify that two *A* instances are not created for the same incoming set of activities (i.e. the intersection between the *previous* instance/s of all *A* instances must be empty).

Table 4.2. Constraints for merge gateways

Merge gateway	Process constraints
 <p>XOR Merge</p>	<ul style="list-style-type: none"> - All <i>A</i> activity instances have as a previous activity instance a completed activity instance of type B_1 or ... or B_n <p>context <i>A</i> inv: <i>previous</i>-><i>size</i>()=1 and <i>previous</i>-><i>exists</i>(<i>b</i> (<i>b.oclIsTypeOf</i>(B_1) or ... or <i>b.oclIsTypeOf</i>(B_n)) and <i>b.status</i>='completed')</p> <ul style="list-style-type: none"> - Each $B_1..B_n$ activity instance is followed by an <i>A</i> activity <p>context Case inv: <i>status</i>='completed' implies <i>activity</i>-><i>select</i>(<i>b</i> <i>b.oclIsTypeOf</i>(B_1) or ... or <i>b.oclIsTypeOf</i>(B_n))-><i>forall</i>(<i>b</i> <i>b.next</i>-><i>exists</i>(<i>a</i> <i>a.oclIsTypeOf</i>(<i>A</i>)))</p>
 <p>OR Merge</p>	<ul style="list-style-type: none"> - An <i>A</i> activity instance must wait for at least an incoming flow <p>context <i>A</i> inv: <i>previous</i>-><i>select</i>(<i>b</i> (<i>b.oclIsTypeOf</i>(B_1) or ... or <i>b.oclIsTypeOf</i>(B_n)) and <i>b.status</i>='completed')-><i>size</i>()>=1</p>
 <p>AND Merge</p>	<ul style="list-style-type: none"> - An activity instance of type <i>A</i> must wait for a set of activities $B_1..B_n$ to be completed <p>context <i>A</i> inv: <i>previous</i>-><i>exists</i>(<i>b</i> <i>b.oclIsTypeOf</i>(B_1) and <i>b.status</i>='completed') and ... and <i>previous</i>-><i>exists</i>(<i>b</i> <i>b.oclIsTypeOf</i>(B_n) and <i>b.status</i>='completed')</p> <ul style="list-style-type: none"> - Each set of completed $B_1..B_n$ activity instances must be related with an <i>A</i> activity instance. <p>context Case inv: <i>status</i>='completed' implies not (<i>activity</i>-><i>exists</i>(<i>b</i> <i>b.oclIsTypeOf</i>(B_1) and <i>b.status</i>='completed' and not <i>b.next</i>-><i>exists</i>(<i>a</i> <i>a.oclIsTypeOf</i>(<i>A</i>)) and ... and <i>activity</i>-><i>exists</i>(<i>b</i> <i>b.oclIsTypeOf</i>(B_n) and <i>b.status</i>='completed' and not <i>b.next</i>-><i>exists</i>(<i>a</i> <i>a.oclIsTypeOf</i>(<i>A</i>)))</p>

4.4. Condition constraints

The sequence flow and the *OR-split* and *XOR-split* gateways may contain condition expressions to control the flow execution at run-time. As an example, Fig. 4.2 shows a conditional sequence flow. In the example, the activity *B* cannot start until *A* is completed and the condition *cond* is satisfied. The condition expression may require accessing the entity types of the domain subschema related to *B* in the workflow-extended model. Through the *Precedes* relationship type, we can also define conditions involving the previous *A* activity instance and/or its related domain information.

To handle these condition expressions we must add, for each condition defined in a sequence flow or in an outgoing link of *OR* and *XOR* gateways, a new constraint over the destination activity. The constraint ensures that the preceding activity satisfies the specified condition, according to the following pattern:

context B inv: previous->forAll(a| a.cond)

Note that these additional constraints only need to hold when the destination activity is created, and thus, they must be defined as *creation-time constraints* [17].

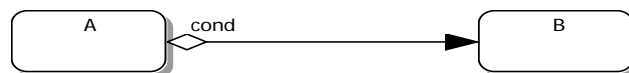


Fig. 4.2. A conditional sequence flow

4.5. Loops

A workflow may contain loops among a group of different activities or within a single activity. In this latter case we distinguish between *standard* loops (where the activity is executed as long as the loop condition holds) and *multi-instance* loops (where the activity is executed a predefined number of times). Every time a loop is iterated a new instance of the activity is created. Fig. 4.3 shows an example of each loop type.

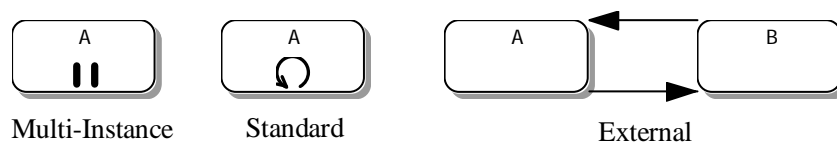


Fig. 4.3. Loop examples

Management of *external loops* does not require new constraints but the addition of a temporal condition in all constraints stating a condition like “an instance of type *B* must be eventually created if an instance of type *A* is completed”. The new temporal condition on those constraints ensures that the *B* instance is created *after* the *A* instance is completed (earlier *B* instances may exist due to previous loop iterations).

Standard loops may be regarded as an alternative representation for conditional sequence flows having the same activity as a source and destination. Therefore, the constraints needed to handle standard loop activities are similar to those required for conditional sequence flows. We need a constraint checking that the previous loop

instance has finished and another one stating that the loop condition is still true when starting the new iteration (again, this is a creation-time constraint). The loop condition is taken from the properties of the activity as defined in the workflow model. Moreover, we need also to check that the activity/ies at the end of the outgoing flows of the loop activity are not started until the loop condition becomes false. To prevent this wrong behavior we should treat all outgoing flows from the loop activity as conditional flows with the condition *'not loopCondition'*. Then, constraints generated to control the conditional flow will prevent next activity/ies to start until the condition *'not loopCondition'* becomes true.

Multi-instance loop activities are repeated a fixed number of times, as defined by the loop condition, which now is evaluated only once during the execution of the case and returns a natural value instead of a boolean value. At the end of the case, the number of instances of the multi-instance activity must be an exact multiple of this value. Assuming that the multi-instance activity is called *A*, the OCL formalization of this constraint would be:

context Case inv: (activity->select(a|a.oclIsTypeOf(A))->size() mod loopCondition)=0

For multi-instance loops the different instances may be created sequentially or in parallel. Besides, we can define when the workflow shall continue. It can be either after each single activity instance is executed (as in a normal sequence flow), after all iterations have been completed (similar to the *AND-merge* gateways), or as soon as a single iteration is completed (similar to the basic *OR-merge* gateway).

4.6. Event management

An event is something that “happens” during the course of the workflow execution. There are three main types of events: *Start*, *Intermediate* and *End* (see Fig. 4.4). A workflow schema may contain several start, intermediate, and end events.

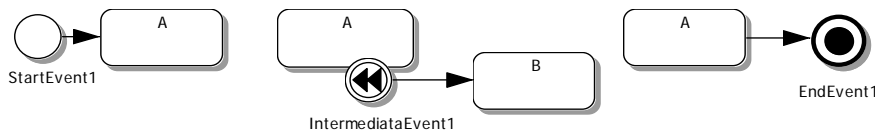


Fig. 4.4 – Examples of events

Start events initiate a new flow, while end events indicate the termination of a flow. Intermediate events are instead used to change the normal flow (for instance, to handle exceptions or to start additional activities). Intermediate events can be attached to an activity (the triggering of the event aborts the activity execution) or can be placed in the middle of a sequence flow between two activities (the flow does not continue until the event is issued).

When a start event is issued, an instance of each activity connected to the event has to start afterwards. Reversely, no activity instance is created in a case before the occurrence of at least a start event. In particular, activity instances for activities connected only to flows coming from one or more start events (as activity *A* in the previ-

ous figure) cannot be created until one of those start events is issued. The formalization of these constraints is the following:

- context Event inv: *eventType.name='StartEvent1' and case.status='completed' implies case.activity->select(a/a.ocIsTypeOf(A) and a.event=self)->size()=1*
- context Case inv: *activity->notEmpty() implies event->exists(e/e.eventType.eventKind='StartEvent')*
- context A inv: *self.event->exists(ev/ ev.eventType.name='StartEvent1')*

For end events defined as *terminate* end events we must add a new constraint stating that no activity instances can be created in the case after the event has been issued. Assuming that *EndEvent1* (Fig. 4.4) is defined as a terminate event, the following constraint must be added to the workflow-extended model:

context Event inv: eventType.name='EndEvent1' implies case.activity->forall (a/ a.start < eventTime)

For intermediate events, the target activity of the event must be executed after the triggering of the event (and it cannot be executed otherwise). Depending on the kind of intermediate event, the interrupted activity will change its status to cancelled or aborted (which, for instance, may prevent the next activity in the normal sequence flow to be started).

The following process constraints are generated for the *IntermediateEvent1* example previously described:

- context Event inv: *eventType.name='IntermediateEvent1' and case.status='completed' implies case.activity->exists(a/a.ocIsTypeOf(B))*
- context Case inv: *activity->exists(a/ a.ocIsTypeOf(B)) implies event->exists(e/e.eventType.name='IntermediateEvent1')*

Obviously, this last constraint is true as long as *B* has no other incoming flows. Otherwise, all incoming flows form an implicit XOR-Merge over *B* and we should generate the constraints according to the pattern for XOR-Merge gateways.

4.7. Applying the translation patterns

As an example, Table 4.3 summarizes the process constraints resulting from applying the translation over the workflow schema of Fig. 2.1.

For sake of brevity, we do not include here the complete set of constraints, but we exemplify in Table 4.4 the full definition of the constraints involved in the *Provide Quotation* activity (the rest of the specifications can be found in the Appendix). The *Provide Quotation* activity involves a set of constraints due to the sequence constraint with *Ask Quotation* activity and a set due to the subsequent XOR split.

Table 4.3. Process constraints for the workflow running example

Activity	Constraints
Ask Quotation	- When the activity instance comes after a <i>Provide Quotation</i> , the latter must have been completed (a single new ask quotation activity can be generated). Otherwise, it must have been created in response to the occurrence of a start event (due to the implicit XOR merge gateway corresponding to the two incoming arrows).

<i>Provide Quotation</i>	<ul style="list-style-type: none"> - A quotation cannot be provided until the <i>Ask Quotation</i> activity has finished. Moreover, if an instance of <i>Ask Quotation</i> is completed, a single <i>Provide Quotation</i> instance must eventually be created - After providing a quotation we can either ask for a new quotation or submit an order, but not both. At least one of them must be executed.
<i>Submit Order</i>	<ul style="list-style-type: none"> - The previous <i>Provide Quotation</i> activity must be completed. Besides, only a single <i>Submit Order</i> instance must be created for the same <i>Provided Quotation</i> instance - After submitting an order, both the <i>Choose Shipment</i> and the <i>Process OrderLine</i> activities must be executed
<i>Choose Shipment</i>	<ul style="list-style-type: none"> - The preceding <i>Submit Order</i> activity instance must be completed. Besides, a single <i>Choose Shipment</i> activity must be executed for each <i>Submit Order</i> activity instance
<i>Process OrderLine</i>	<ul style="list-style-type: none"> - The preceding <i>Submit Order</i> activity must be completed - The system must exactly execute as many <i>Process OrderLine</i> activity instances as the number of order (quotation) lines for the related order
<i>Ship Order</i>	<ul style="list-style-type: none"> - The order cannot be shipped until the shipment has been chosen and all order lines have been processed. Then, a <i>Ship Order</i> activity instance must be executed before ending the case
<i>Pay Order</i>	<ul style="list-style-type: none"> - An order cannot be paid until it has been shipped. A single <i>pay order</i> activity shall be created in response to each order shipment

5. Code-generation of the workflow-extended domain model

A workflow-extended domain model is a completely standard domain model. No new modeling primitives have been created to express the extension of the original model with the required workflow information. Therefore, any method or tool able to provide an automatic implementation of the initial domain model can also cope with the automatic generation of our workflow-extended model in any final technology platform using general-purpose MDD techniques and frameworks.

For instance, activity classes (as *AskQuotation* or *ProvideQuotation*) could be implemented as database tables or Java classes while process constraints could be implemented as triggers and method preconditions respectively. Note that a translation from OCL into SQL or Java is already provided by several tools (e.g., [9], [14]), covering also efficient implementation of OCL constraints [6].

Table 4.4. Constraint definitions for the Provide Quotation activity

Constraints due to the sequence with Ask Quotation	The preceding activity must be of type <i>Ask Quotation</i> and must be completed
	<i>context ProvideQuotation inv: previous->size()=1 and previous->exists(a/a.ooclIsTypeOf(AskQuotation) and a.status='completed')</i>
	No two instances may be related with the same <i>Ask Quotation</i> instance
	<i>context ProvideQuotation inv: ProvideQuotation.allInstances()->is-Unique(previous)</i>
	A Provide Quotation instance must exist for each completed <i>Ask Quotation</i>

	<i>context Case inv: status='completed' implies activity-> select(a a.oclIsTypeOf(AskQuotation) and a.status='completed')->forall(a a.next->exists(b b.oclIsTypeOf(ProvideQuotation) and a.end<=b.start) and a.next->size()=1)</i>
Constraints due to the XOR split	The next activity must be either another <i>Ask Quotation</i> instance or a <i>Submit Order</i> instance, but not both
	<i>context ProvideQuotation inv: next->select (a a.oclIsTypeOf(AskQuotation) or a.oclIsTypeOf(ProvideQuotation))->size()<=1</i>
	If the <i>Provide Quotation</i> instance is completed, an <i>Ask Quotation</i> or a <i>Submit Order</i> must be created before ending the case.
	<i>context Case inv: status='completed' implies activity->select(a a.oclIsTypeOf(ProvideQuotation) and a.status='completed')-> forall (a a.next-> exists(b b.oclIsTypeOf(AskQuotation) or b.oclIsTypeOf(SubmitOrder)))</i>
	Only <i>Ask Quotation</i> activity instances or <i>Submit Order</i> instances may follow a <i>Provide Quotation</i> instance
	<i>context ProvideQuotation inv: next->forall(b b.oclIsTypeOf(AskQuotation) or b.oclIsTypeOf(SubmitOrder)</i>

6. Related work

Research on business process in software engineering has mainly addressed the correctness of the design of the workflow model (see [11] as an example) or its direct implementation in specific final technology platforms (see [2] for an implementation over a relational database and [4] for an implementation using web technologies). Integration of workflows and MDD approaches has only been explored from a general framework perspective [12].

As far as we know, ours is the first proposal where both workflow information and process constraints are automatically derived from a workflow model and integrated within a platform-independent domain model. As we have seen in the previous section, this integration permits to generate workflow applications in any final technology without requiring to develop an specific treatment for the workflow model.

Moreover, ours is also the first translation of a workflow model into a set of OCL declarative constraints. Such a translation is necessary regardless how these constraints are to be enforced in the final workflow implementation.

Very few examples of translations to other declarative languages exist (e.g., see [3] for a translation to LTL temporal logics). In literature, workflow metadata and OCL constraints have only been used in [10] to manually specify workflow access control constraints and derive authorization rules, in [1] to express constraints with respect to the distribution of work to teams, in ArgoUWE [15] to check for well-formedness in the design of process models, in [20] to manually specify business models with UML and in [16] to specify the contracts for the transformation of activity diagrams into BPEL4WS.

7. Conclusions

In this paper we presented an automatic approach to integrate the semantics of business process specifications within domain models.

Once the designer has specified both the workflow and the domain models separately, we build an integrated workflow-extended domain model by means of adding to the domain model *(i)* the definition of a set of new entity and relationship types for workflow status tracking and *(ii)* the rules for generating the integrity constraints on such types, needed for enforcing the business process specification.

The integration of both the domain and the workflow aspects in a single extended domain model permits a homogeneous treatment of the workflow-based application. For instance, we can apply the usual model-driven development methods over our extended model to generate its automatic implementation in any technology platform.

To make the proposed approach viable, we have developed a visual editor prototype that allows to design BPMN diagrams (see the tool of Fig. 2.1) and to automatically generate the corresponding workflow subschema (Fig. 3.1) and its process constraints, according to the guidelines presented in this paper. In particular, given the XML representation of the workflow model and the XMI representation of the initial domain model (in particular the XMI version used by *MagicDraw*), our tool generates a new XMI file containing the workflow-extended model and the process constraints.

Future work will include the extension of our translation patterns to directly cover the full expressivity of the BPMN notation and the study and comparison of different implementation options for the workflow-extended models depending on application-specific requirements. Also, we would like to explore the possibility of using our extended model as a bridge to facilitate reverse-engineering of existing applications into their original workflow models and to ease keeping them aligned. Finally, we plan to develop a method that, from the generated process constraints, is able to compute the list of activities that can be enacted by a user in a given case (i.e. those activities that can be created without violating any of the workflow constraints according to the case state at that specific time).

Acknowledgments

This work has been partially supported by the Italian grant FAR N. 4412/ICT, the Spanish-Italian integrated action HI2006-0208, the grant BE 00062 (Catalan Government) and the Spanish Research Project TIN2005-06053.

References

1. Aalst, W. M. P. v. d., Kumar, A.: A reference model for team-enabled workflow management systems. *Data & Knowledge Engineering* 38 (2001) 335-363
2. Bae, J., Bae, H., Kang, S.-H., Kim, Y.: Automatic Control of Workflow Processes Using ECA Rules. *IEEE Transactions on Knowledge and Data Engineering* 16 (2004) 1010-1023
3. Brambilla, M., Deutsch, A., Sui, L., Vianu, V.: The Role of Visual Tools in a Web Application Design and Verification Framework: a Visual Notation for LTL Formu-

- lae. In: Proc. 5th Int. Conf. in Web Engineering (ICWE'05), LNCS, 3579 (2005) 557-568
4. Brambilla, M., Ceri, S., Fraternali, P., Manolescu, I.: Process Modeling in Web Applications. ACM Transactions on Software Engineering and Methodology 15 (2006) 360-409
 5. Cabot, J., Raventós, R.: Conceptual Modelling Patterns for Roles. Journal on Data Semantics V (2006) 158-184
 6. Cabot, J., Teniente, E.: Incremental Evaluation of OCL Constraints. In: Proc. 18th Int. Conf. on Advanced Information Systems Engineering, LNCS, 4001 (2006) 81-95
 7. Combi, C., Pozzi, G.: Temporal Conceptual Modelling of Workflows. In: Proc. 22nd Int. Conference on Conceptual Modeling (ER'03), LNCS, 2813 (2003) 59-76
 8. Costal, D., Gómez, C., Queralt, A., Raventós, R., Teniente, E.: Facilitating the definition of general constraints in UML. In: Proc. 9th Int. Conf on Model Driven Engineering Languages and Systems (MODELS'06), LNCS, 4199 (2006) 260-274
 9. Demuth, B., Hussmann, H., Loecher, S.: OCL as a Specification Language for Business Rules in Database Applications. In: Proc. 4th Int. Conf. on the Unified Modeling Language (UML 2001), LNCS, 2185 (2001) 104-117
 10. Domingos, D., Rito-Silva, A., Veiga, P.: Workflow Access Control from a Business Perspective. In: Proc. ICEIS, 3 (2004) 18-25
 11. Eshuis, R., Wieringa, R.: Verification support for workflow design with UML activity graphs. In: Proc. 22rd Int. Conf. on Software Engineering (ICSE'02), (2002) 166-176
 12. Hur, W., Jung, J.-y., Kim, H., Kang, S.-H.: Model-Driven Approach to workflow execution. In: Proc. BPM'04, LNCS, 2080 (2004) 261-273
 13. IBM. WebSphere MQ Workflow.
<http://www.ibm.com/software/ts/mqseries/workflow/v332/>
 14. KlasseObjecten. Octopus: OCL Tool for Precise Uml Specifications.
<http://www.klasse.nl/octopus/index.html>
 15. Knapp, A., Koch, N., Zhang, G., Hassler, H.: Modeling Business Processes in Web Applications with ArgoUWE. In: Proc. UML'04, LNCS, 3273 (2004) 69-83
 16. Koehler, J., Hauser, R., Sendall, S., Wahler, M.: Declarative techniques for model-driven business process integration. IBM Systems Journal 44 (2005) 47-65
 17. Olivé, A.: A method for the definition of integrity constraints in object-oriented conceptual modeling languages. Data & Knowledge Engineering 58 (2006) 243-262
 18. OMG/BPMI: Business Process Management Notation v.1. OMG Adopted Specification (2006)
 19. Oracle. Workflow 11i.
<http://www.oracle.com/appsnet/technology/products/docs/workflow.html>
 20. Takemura, T., Tamai, T.: Rigorous Business Process Modeling with OCL. In: Proc. OCL Workshop in MODELS'06, (2006)
 21. White, S. A.: Process Modeling Notations and Workflow Patterns. BPTrends (2004)
 22. Wynn, M. T., Edmond, D., Aalst, W. M. P. v. d., Hofstede, A. H. M. t.: Achieving a general, formal and decidable approach to the OR-join in Workflow using Reset nets. In: Proc. 26th Int. Conf. on Application and Theory of Petri Nets (ICATPN'06), LNCS, 3536 (2005) 423-443

Appendix

The application of the translation patterns over the workflow schema of Figure 2.1 produces the following set of process constraints. Constraints are grouped according to the main activity they affect. For each constraint we also indicate the workflow construct that generates the constraint.

Apart from the constraints specific for each activity, all activity instances must not start before the occurrence of a start event or after the occurrence of a *terminate* end event, as already seen in Section 4.6.

Ask Quotation activity

- Constraints due to the start event
 1. A single *Ask Quotation* activity instance must eventually exist for each issued *Start* event
context Event inv: eventType.name='Start' and case.status='completed' implies case.activity->select(a| a.ocIsTypeOf(AskQuotation) and a.event=self)->size()=1
- Constraints due to the implicit *XOR-Merge* between the incoming flows from the *Start* event and from the outgoing flow of the *XOR-split* after *Provide Quotation*
 1. All *Ask Quotation* instances must be related either with a complete *Provide Quotation* instance or with an issued *Start* event.
context AskQuotation inv: previous->notEmpty() implies previous->size()=1 and previous->exists(a| a.status='completed' and a.ocIsTypeOf(ProvideQuotation))
context AskQuotation inv: previous->isEmpty() implies event-> exists(ev| ev.eventType.name='Start')
 2. No two instances of *Ask Quotation* related with the same *Provide Quotation* instance may exist. Note that when we iterate over the loop between *Ask Quotation* and *Provide Quotation* activities, new activity instances are generated in each iteration.
context AskQuotation inv: AskQuotation.allInstances()-> isUnique(previous)

Provide Quotation activity

- Constraints due to the sequence constraint with *Ask Quotation* activity
 1. The preceding activity must be of type *Ask Quotation* and must be completed
context ProvideQuotation inv: previous->size()=1 and previous->exists(a| a.ocIsTypeOf(AskQuotation) and a.status='completed')
 2. No two instances may be related with the same *Ask Quotation* instance
context ProvideQuotation inv: ProvideQuotation.allInstances()-> isUnique(previous)
 3. A *Provide Quotation* instance must exist for each completed *Ask Quotation*
context Case inv: status='completed' implies activity->select(a| a.status='completed' and a.ocIsTypeOf(AskQuotation))->

*forall(a/a.next->exists(b/ a.end<=b.start and
b.ocllsTypeOf(ProvideQuotation)) and a.next->size()=1)*

- Constraints due to the *XOR-split*
 1. The next activity must be either another Ask Quotation instance or a Submit Order instance, but not both
*context ProvideQuotation inv: next->select(a/ a.ocllsTypeOf(AskQuotation)
or a.ocllsTypeOf(ProvideQuotation))-> size()<=1*
 2. If the Provide Quotation instance is completed, an Ask Quotation or a Submit Order must be created before ending the case
*context Case inv: status='completed' implies activity->select(a/
a.status='completed' and a.ocllsTypeOf(ProvideQuotation))->
forall (a/ a.next-> exists(b/ (b.ocllsTypeOf(AskQuotation) or
b.ocllsTypeOf(SubmitOrder)) and b.start>=a.end))*
 3. Only AskQuotation activity instances or Submit Order instances may follow a Provide Quotation instance
*context ProvideQuotation inv: next->forall(b/ b.ocllsTypeOf(AskQuotation)
or b.ocllsTypeOf(SubmitOrder))*

Submit order activity

- Constraints due to outgoing flow of the *Provide Quotation XOR-split*
 1. The previous activity must be of type *Provide Quotation* and must be completed
*context SubmitOrder inv: previous->size()=1 and previous->exists(a/
a.status='completed' and a.ocllsTypeOf(ProvideQuotation))*
 2. No two instances of Submit Order may be related with the same provide quotation instance
context SubmitOrder inv: SubmitOrder.allInstances()-> isUnique(previous)
- Constraints due to *AND-split* between *Choose Shipment* and *Process Orderline*
 1. For each *Submit Order*, the *Choose Shipment* and the *Process OrderLine* activities must be executed
*context Case inv: status='completed' implies activity->select(a/
a.status='completed' and a.ocllsTypeOf(SubmitOrder))-> forall(a/a.next->
exists(b/ b.ocllsTypeOf(ChooseShipment)) and a.next->exists(b/
b.ocllsTypeOf(ProcessOrderLine)))*
 2. Only *Choose Shipment* activity instances or *Process OrderLine* instances may follow a *Submit Order* instance
*context SubmitOrder inv: next->forall(b/ b.ocllsTypeOf(ChooseShipment)
or b.ocllsTypeOf(ProcessOrderLine))*

Choose Shipment

- Constraints due to the outgoing flow of the submit order *AND-split*
 1. The previous activity must be of type *SubmitOrder* and must be completed
*context ChooseShipment inv: previous->size()=1 and previous->exists(a/
a.ocllsTypeOf(SubmitOrder) and a.status='completed')*

2. No two instances of *Choose Shipment* may be related with the same *Submit Order*
context ChooseShipment inv:
ChooseShipment.allInstances()-> isUnique(previous)

Process OrderLine

- Constraints due to the outgoing flow of the *Submit Order AND-split*
 1. The previous activity must be of type *SubmitOrder* and must be completed
context ProcessOrderLine inv: previous->size()==1 and previous->exists(a| a.oclIsTypeOf(SubmitOrder) and a.status='completed')
 2. No two instances of *Process OrderLine* may be related with the same submit order
context ProcessOrderline inv:
ProcessOrderline.allInstances()-> isUnique(previous)
- Constraints due to the multi-instance loop
 1. There must exist a *Process OrderLine* instance for each *OrderLine* of the order related with the activity
context Case inv: (activity->select(a| a.oclIsTypeOf(ProcessOrderLine))->size()) mod (ProcessOrderLine.allInstances()->size()) = 0

Ship Order

- Constraints due to the *AND-Merge*
 1. We cannot start shipping the order until the activities *Choose Shipment* and *Process OrderLine* have been executed. We cannot ship the order until all order lines have been processed, and thus, we must wait for all required *Process OrderLine* instances to be completed.
context ShipOrder inv: previous->exists(b| b.oclIsTypeOf(ChooseShipment) and b.status='completed') and previous->select(b| b.oclIsTypeOf(ProcessOrderLine) and b.status='completed')->size()==self.order.quotation.orderLines->size()
 2. A *Ship Order* instance must eventually exist if the *Choose Shipment* and *Process OrderLine* activities have been issued.
context Case inv: status='completed' implies not (activity->exists(b| b.oclIsTypeOf(ChooseShipment) and b.status='completed' and not b.next->exists(a| a.oclIsTypeOf(ShipOrder))) and activity->exists(b| b.oclIsTypeOf(ProcessOrderLine) and b.status='completed' and not b.next->exists(a| a.oclIsTypeOf(ShipOrder))))
 3. The previous instances of two different *Ship Orders* must have an empty intersection.
context ShipOrder inv: ShipOrder.allInstances()->forall(s1,s2| s1<>s2 implies s1.previous->intersection(s2.previous)-> isEmpty())

Pay Order

- Constraints due to the sequence constraint with *Ship Order* activity
 1. The previous activity must be of type *Ship Order* and must be completed
context PayOrder inv: previous->size()=1 and previous->exists(a/ a.status='completed' and a.oclIsTypeOf(ShipOrder))
 2. No two instances may be related with the same *Ship Order* instance
context PayOrder inv: PayOrder.allInstances()-> isUnique(previous)
 3. A *Pay Order* instance must eventually exist for each completed *Ship Order*
context Case inv: status='completed' implies self.activity-> select(a/ a.oclIsTypeOf(ShipOrder) and a.status='completed')->forAll(a/a.next->exists(b/b.oclIsTypeOf(PayOrder)))