Extending Conceptual Schemas with Business Process Information

Marco Brambilla[1], Jordi Cabot[2], Sara Comai[1]

[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). Typically, workflow models are managed through specific subsystems, called workflow management systems, to ensure a consistent behaviour of the applications with respect to the organization business process. However, for small organizations and/or simple business processes, the complexity and capabilities of these dedicated workflow engines may be overwhelming. In this paper we therefore advocate for a different and lightweight approach, consisting in the integration of the business process specification within the system conceptual schema. We show how a workflow-extended conceptual schema can be automatically obtained, which serves both to enforce the organization business process and to manage all its relevant domain data in a unified way. This extended model can be directly processed with current CASE tools, for instance, to generate an implementation of the system (including its business process) in any technological platform.

Keywords: business process models, BPMN, integrity constraints, UML, OCL

1. Introduction

All software systems must include a formal representation of the *knowledge* of the domain. In conceptual modeling, this representation is known as the *conceptual schema* of the software system [22]. However, software development processes for complex business applications usually require the additional definition of a workflow model to express logical precedence and process constraints among the different business activities (i.e., the units of work).

Workflow models are usually specified through dedicated languages (e.g., Business Process Management Notation - BPMN [35]) and implemented with the help of specialized workflow management systems (WFMSs), e.g., see [36] [21], which are heavy-weight applications focused on the control aspects of the business process enactment. This is clearly the best option to manage large workflow models. However, in some cases organizations may prefer a more lightweight approach that does not require acquiring a specific workflow subsystem.

This paper tackles the problem of defining a light-weight approach to the implementation of business processes within software applications, without the use of specialized WFMSs, which represents a relevant issue in several application scenarios. Indeed. alternative solutions to complete WFMSs can be preferred in case of simple business requirements, small organizations, or when the business process needs are going to be drowned into a larger system that is being implemented *ad hoc* for the organization. In these cases, designing and implementing the workflow using the same methods, notations and tools used to develop the rest of the system can be convenient and cost effective for the organization.

Along these development lines, some approaches have focused on the implementation of workflow models in specific technology platforms, as relational databases (generally in the form of triggers [3]), Web applications (by means of hypertextual links and buttons properly placed in Web pages, thus restricting the user navigation [9]), or Web services (through transformation into Business Process Execution Language for Web Services - BPEL4WS [25] specifications). This way, the workflow definition becomes part of the system implementation and no specific workflow engine is required. However, these approaches can be hardly generalized to technologies different from the ones for which they have been conceived (e.g., to new technology platforms), make difficult a wider adoption of business processes within the organizations, and present some limitations regarding the supported expressivity for the initial workflow model and/or its integration with the conceptual schema.

As an alternative, in this paper we propose a formalized model-driven development (MDD) approach for developing workflow-based applications and advocate for the automatic integration of the workflow model within the (platform-independent) conceptual schema. The resulting workflow-extended conceptual schema includes in a single schema both the business process specifications and the domain knowledge, providing a *unified view* of the system and allowing treating both dimensions in a homogeneous way when implementing, verifying, and evolving the system. The integration is done at the model level. Therefore, current modeling tools can be used to manage our workflow extended schema, no matter the target technology platform or the purpose of the tool (e.g. verification, code-generation, …).

The rest of the paper is structured as follows: Sections 2 summarizes and motivates our approach and its advantages. In sections 3 and 4 the conceptual schema, the workflow concepts, and our case study are illustrated. Section 5 introduces the normalization phase. In Sections 6 and 7 we provide the definition of the workflow-extended conceptual schema and of the OCL (Object Constraint Language [32]) process constraints, respectively. Section 8 sketches possible implementation strategies for this extended model. Section 9 portrays our prototype tool implementation. Then, Section 10 compares our approach with related work and in Section 11 we draw our conclusions and discuss future work.

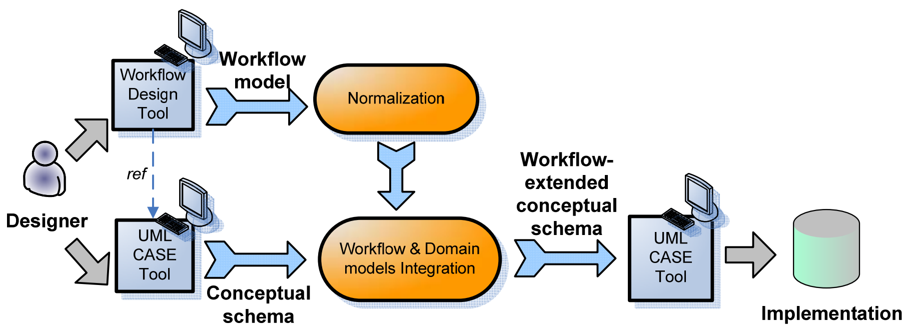


Fig. 1. MDD process for workflow-based applications

2. Overview of the proposed approach and of its benefits

Our MDD approach for developing workflow-based applications is sketched in Fig. 1: the designer specifies the conceptual schema (e.g., in UML) and the workflow model of the application (e.g., in BPMN), using the appropriate design tools. At this stage some links between the workflow model and the conceptual schema can be already identified. Typically, they represent the usage relationship that associates objects of the application domain to activities in the workflow model.

The workflow model may need a normalization transformation for homogenizing the notation and making it fit for the next (automatic) steps.

The conceptual schema and the workflow model undergo to the integration transformation phase that produces the *workflow-extended conceptual schema*. More specifically, given a conceptual schema *c* and a workflow model *w*, it is possible to automatically derive a full fledged conceptual schema *c'* 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. Several workflow models can be integrated with the same conceptual schema since the process constraints of each workflow model do not interfere. This is guaranteed by the construction process of the workflow-extended model. This extended schema can then be managed using any commercial UML CASE tool.

The whole approach has been implemented in a prototype tool that automatically translates the workflow specifications into a set of types and constraints on the conceptual schema, according to a set of translation patterns described in the paper.

The focus of the paper will be on the platform-independent transformations of the conceptual models; however, some ideas on how to implement the workflow-extended conceptual schema into target platforms will be provided. As reference models, throughout the paper we will use UML class diagrams for the representation of conceptual schemas and OCL constraints to represent the process constraints. For the workflow, we will adopt a particular business process notation, namely BPMN [35], for sake of readability and concreteness. Indeed, business analysts are well aware of business process modeling techniques but are not so familiar with software engineering notations and practices. Recently, BPMN and other domain-specific notation have been increasingly accepted in the field, thus we based our examples on a notation that business roles in the enterprises are familiar with.

Our model transformations are based on the concepts and definitions specified by BPDM (Business Process Definition Metamodel [34]), a platform- and notation-independent metamodel for defining business processes. Since BPDM is a common metamodel for all business process notations (e.g., it includes all concepts of BPMN and UML activity diagrams), our approach can be used exactly in the same way when using activity diagrams or any other BPDM-compliant notation to model the workflows. The proposed approach is therefore general-purpose and is valid regardless of the adopted business process notation.

2.1. *Motivation and discussion*

The main advantage of the proposed approach is that the workflow-extended conceptual schema includes in a single conceptual schema both the business process specifications and the domain knowledge. Since the workflow-extended model is automatically generated from the workflow model and the conceptual schema, a *unified view* of the system is hence available without any additional effort by the designer. This allows treating both dimensions in the resulting model in a homogeneous and consistent way when implementing, verifying, and evolving the system. Thanks to this unified view, our workflow-extended schemas enable the definition of more expressive business constraints, generally not allowed by common business process definition languages such as timing conditions [13] or conditions involving both business process and domain information.

Moreover, since the integration of the workflow and conceptual schemas is done at the model level, the resulting workflow-extended conceptual schema is a *platform-independent* model. Thanks to the current state of the art of model-to-model and model-to-text transformation tools, integrating different notations in the same approach (e.g., UML class diagrams, OCL, and BPMN) does not make a difference. Indeed, the extraction and integration process will simply consider models conforming to different metamodels (e.g., UML and BPDM). Anyway, the model transformations involved are straightforward and compliant with the MDD approach.

Once the final workflow-extended schema is produced, it can benefit from any method or tool designed for managing a generic conceptual schema, 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 the workflow information in our workflow-extended schema, since it is a completely standard UML model [30]. In this sense, with our approach we do not need to develop specific techniques for workflow models nor to use specific tools for managing them.

Finally, once (automatically) implemented (with the help of any of the current UML/OCL CASE tools offering code-generation capabilities), the workflow-extended conceptual schema 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 executed according to the initial business process specification.

2.2. *Original contributions of the paper*

To our knowledge, ours is the first approach that automatically derives a platform-independent conceptual schema integrating both domain and business process information in a unified view. A first version of this proposal has been published in [7]; however, this paper extends [7] in several directions. In particular, the main original contributions of this paper include:

- The introduction of a normalization phase to simplify the initial workflow models and extend the set of workflow patterns we can directly cover with our method.

- A complete description of the process that allows obtaining the workflow-extended conceptual schema starting from the domain model and the workflow model.

- An extended and refined version of the translation of process constraints, including also the management of the start, end, and intermediate events in the business process specification. Such events can represent different event types (message, exception, rule, timer, and so on).

- The specification of different integration scenarios that can be used in the transformation process and a discussion on their trade-offs in terms of the complexity of the resulting extended schema and of the process constraints.

- The description of different implementation alternatives for the workflow extended schema towards target platforms.

- The description of our tool implementation, supporting all the (automatic) model transformations.

3. Conceptual schemas

A *conceptual schema* (also known as *domain model*) defines the knowledge about the domain that an information system must have to perform its business functions. Without loss of generality, we will represent conceptual schemas using UML [30].

The most basic constructors in conceptual schemas are entity types (i.e., classes in the UML terminology), relationship types (i.e., associations) and generalizations.

An *entity type E* describes the common characteristics of a set of entities (i.e., objects) of the domain. Each entity type *E* may contain a set of *attributes*.

A binary *relationship type R* has a name and two participants. A *participant* is an entity type that plays a certain role in the relationship type. Each relationship (i.e., link) between the two participants represents a semantic connection between the entities. A participant in *R* may have a minimum and maximum cardinality. The minimum cardinality *min* between participants p_1 and p_2 in *R* indicates that all entities of E_1 (type of the participant p_1) must be related at least with *min* entities of E_2 (type of the participant p_2). A maximum cardinality *max* between p_1 and p_2 in *R* defines that entities of E_1 cannot be related with more than *max* entities of E_2.

A *generalization* is a taxonomic relationship between a more general entity type *E* (supertype) and a set of more specific entity types $E_1,\ldots,E_n$ (subtypes).

As an example, Fig. 2 shows a conceptual schema, represented in UML, meant to (partially) model a simple e-commerce application. It consists of the entity types *Product*, *Quotation*, *QuotationLine* (to record the details of the products included in the quotation), and *Order* (an order is generated by each quotation accepted by the customer, and then, its quotation lines are referred to as order lines). According to the cardinality constraints in the relationship types, all quotation must include at least one product and orders must be of a single quotation.

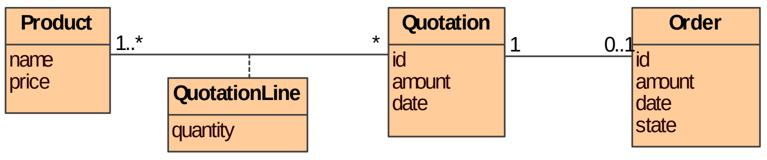


Fig. 2. A partial conceptual schema for an e-commerce application

4. Business Processes 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, the Business Process Definition Metamodel [34] (BPDM), and the Business Process Management Notation [35] (BPMN).

BPDM is a standard proposed by OMG for representing and modeling business processes independent of any notation or methodology. This is done by proposing a unified metamodel that captures the common meaning behind the different notations and technologies. The metamodel is a MOF-compliant [33] metamodel. As such, BPDM also defines a XML syntax for storing and transferring business process models between tools and infrastructures. BPDM has been evaluated in [27] as the best business process interchange format in terms of expressivity.

BPMN perfectly fits with the BPDM metamodel and provides a graphical notation to express BPDM business processes. However, the specification of the business process can be provided with any other notation or language, including UML Activity Diagrams [30]. Several works evaluated and compared the different notations for specifying business processes (e.g., see [26], [27], [40], [2]), highlighting strengths and weaknesses of every proposal. The results of our approach using one of these alternative notations would be quite similar. Indeed, our approach can be directly applied to any specification compliant with BPDM.

In our work, we focus on the core part of the BPDM metamodel. 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.

In the sequel, we will exemplify the proposed approach on a case study consisting of a simplified purchase process, illustrated using the BPMN notation in Fig. 3.

According to the BPDM semantics, the depicted diagram specifies a process involving two actors (represented by the two swimlanes): a customer and a seller. The customer starts the process 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 (*Change quotation* activity, followed by the repetition of the *Provide quotation* activity) or to accept it (then the order is submitted). For simplicity, it is not modeled what happens if they never reach an agreement. Depending on the complexity of the order, the process can follow two alternative paths: the first consists only of a *Standard Shipment* activity, while the second requires the customer to specify the kind of shipment he prefers (*Choose shipment*). After the choice, the Seller takes the order in charge and performs two parallel activities: the arrangement of the transport plan and the processing of each order line. The latter is represented by the multi-instance activity called *Process order line*: 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 defined (i.e., after the AND merge synchronization), the path reaches the join point with the alternative path of the standard shipment. Independently on the kind of shipment, the *Ship order* activity is performed, and then two uncontrolled branches take place: the customer receives the goods and the seller issues and sends the invoice. When both activities have completed (synchronization AND gateway), the user pays for the goods, and thus closes the process.

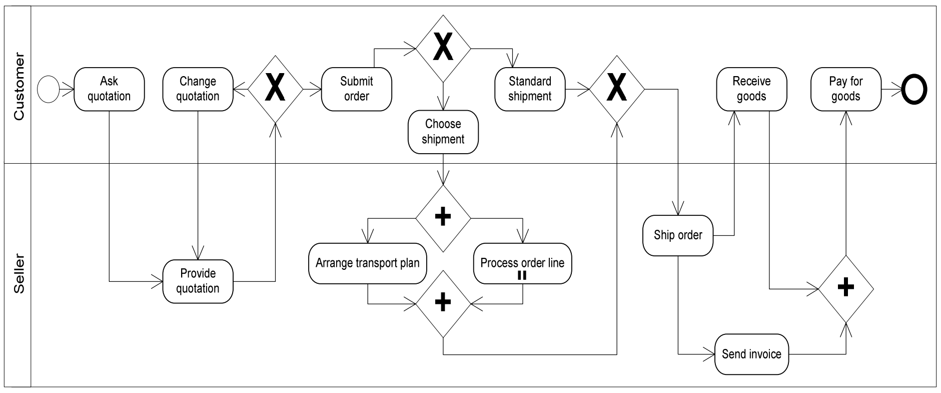


Fig. 3. Example of a workflow model

5. Normalization Phase

Before addressing the actual integration of the workflow model and the conceptual schema, the business process specification usually needs to be normalized. This step simplifies the processing of the workflow model later on without losing generality in the coverage of the business process specification admitted in our method.

Workflow languages allow different equivalent representations of the same business semantics (see [35] for details) and define several complex constructs that can be derived from more basic ones. The normalization phase tackles these problems by applying a set of model to model transformations that ensure a coherent representation and render all the complex concepts in terms of simple ones. Notice that this phase does not aim at the reconciliation of different business processes. Instead, it aims at unifoming the notation of different design styles that could be adopted even within the same notation. The main issues addressed in this phase are:

- *Nested structures:* if the business process is specified by means of nested subprocesses, they are flattened into a single-level business process that includes all tasks that were included in the subprocesses. If the subprocess contained only one lane, all the activities are moved inside the current lane of the main process; if more lanes were contained in the subprocess, they are transferred to the current pool of the main process, together with their respective activities, thus introducing new lanes in the flattened process.
- *Different notation styles:* all different notations with the same BPDM semantics are homogenized in a single BPMN notation style (some examples are shown next). Thanks to this step, the business process will use only a single representation for each modeled behavior.
- *Concatenation of gateways:* if two or more gateways are directly connected by a control flow, the transformation adds a fake intermediate activity in the middle of every gateway pair. This simplifies the integration job, since it permits to work in a modular fashion when generating the constraints and the rules imposed by the gateways. Fake activities can be treated as activities that can be immediately en-

acted as soon as their process constraints are satisfied, and then automatically completed without neither user interaction nor business action execution.

Fig. 4 shows the result of applying the normalization phase on the workflow model specified in Fig 3. The elements added because of the normalization are highlighted in color and bold line face. Only the last two transformations apply to this example. To avoid the alternative notation for XOR-merge (consisting in directly connecting two incoming arrows to an activity), the XOR-merge after the *Ask quotation* activity is made explicit and added to the model; analogously, to avoid two outgoing arrows from the *Ship order* activity, an AND-spit gateway is added. To remove the configurations of two connected gateways, a fake activity is added (*EmptyActivity1*).

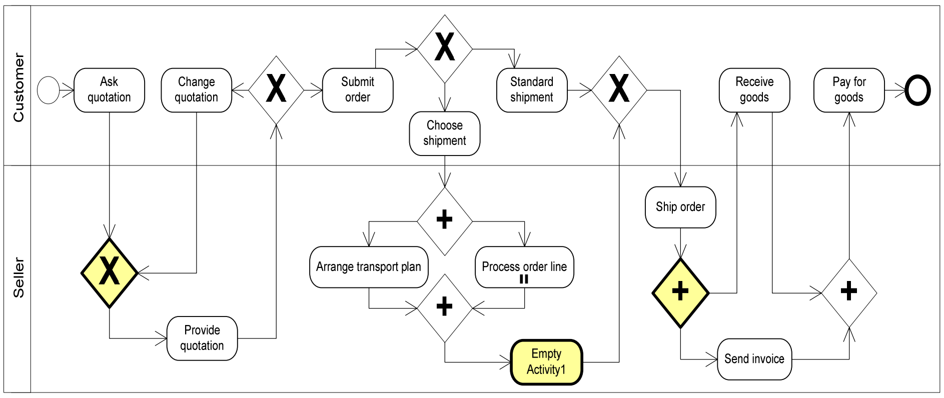


Fig. 4. Example of a normalized workflow schema

6. Extending Conceptual Schemas with Business Process Information

Given an initial conceptual schema *c*, the workflow-extended conceptual schema *c'* of the workflow-based application *w* is obtained by extending *c* with some additional elements derived from the business process specification *w*. We will focus on the case of a single business process; however, our extensions to the conceptual schema suffice when considering different business processes on the same domain. Indeed, in our approach several workflow models can be integrated with the same conceptual schema since the process constraints of each workflow model do not interfere due to the construction process of the workflow-extended model.

6.1. *Generation of the workflow-extended conceptual schema*

The workflow-extended conceptual schema must include: *(i)* the original conceptual schema, *(ii) user*-related information, *(iii) workflow*-related information, *(iv)* a set of possible relationships between the conceptual 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 Section 7). To illustrate the generation of these different parts of the workflow-extended conceptual schema we will use the workflow model of Fig. 34 and we will assume that the initial conceptual schema is the one shown in Fig. 2.

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

(i) *Domain subschema*: All entity types in *E* and their relationships (i.e., associations) and generalizations remain unchanged in the workflow-extended model (bottom part of Fig. 5).

(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. 5): 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. 5) 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 [35]. 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 for executing them. In our case study, *AskQuotation*, *ProvideQuotation*, etc. would be instances of *ActivityType*.

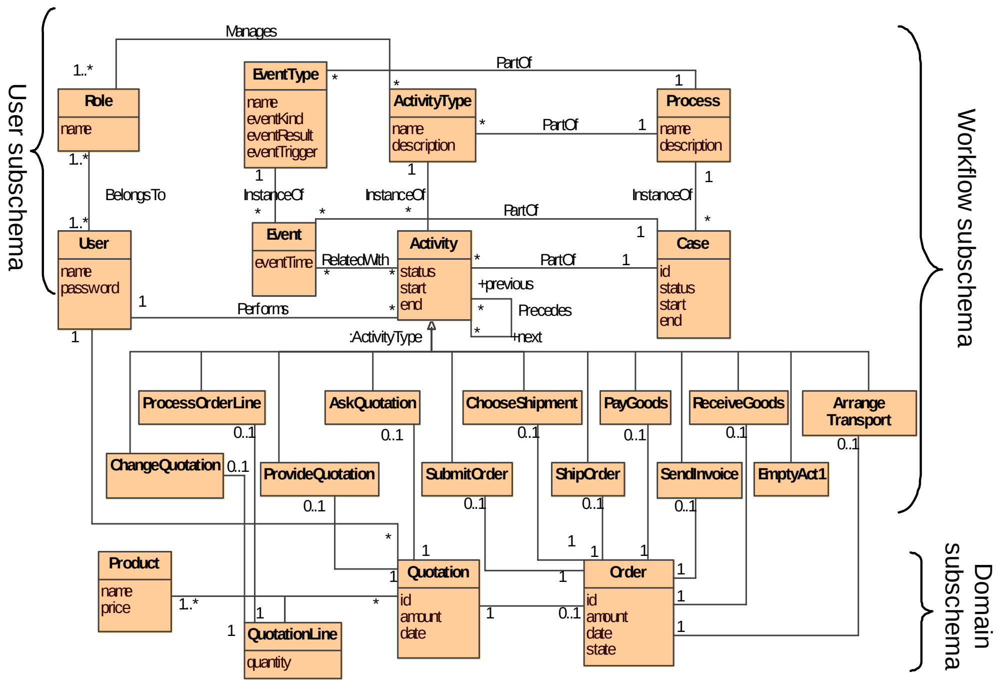


Fig. 5. Workflow-extended conceptual schema

- 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 occurred 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* [30] for this set of generalizations). The name of the subtype is the name of *a* (e.g., in Fig. 5 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 in 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. 5), a set of relationship types are established: *Quotations* are associated with the activities *Ask Quotation* and *Provide Quotation*;

QuotationLines are associated with the *ProcessOrderLine* activity; and *Orders* are associated with the activities *Submit Order*, *Process OrderLine*, *Ship Order*, *Pay for goods* and so forth. 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.

6.2. *Complexity of the workflow-extended conceptual schema*

Clearly, the workflow-extended schema is more complex than the original conceptual schema. However, we believe that this increased complexity is compensated by the fact that it may be automatically generated (with our method) and processed (for instance, with code-generation tools) and thus, the designer does not need to directly manipulate it. Moreover, the size of the extension is 1) constant regardless the size of the initial conceptual schema and 2) linear with respect to the number of activities in the workflow. Therefore, in most cases, the extension will be small when compared with the size of the initial conceptual schema.

We would like to remark that when proposing our workflow-extended schema we opted for balancing the size of the workflow subschema with the complexity of the process constraints. 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 [10]). Similarly, simpler extensions could be used instead but then, as a trade-off, the process constraints would become much more complex. To better illustrate this discussion, two other alternative workflow-extended models are provided in the Appendix A. All three alternatives share the same philosophy and provide the same kind of benefits, and thus, designer may choose any of them when applying our method.

7. 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 into constraints over the population of the $s_{a1},\ldots,s_{am}$ 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 [32]. 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. For instance, a constraint like:

context A inv: condition

implies that all instances of the type *A* must verify *condition*.

Constraints are defined to restrict only the execution of the workflow they are created for (the context type of the constraints is always a specific activity and not an entity type of the domain subschema). Therefore, no interferences among different workflows occur, even if they are defined over an overlapping subset of the conceptual schema.

Even though some of the constraints may seem quite complex, we would like to remark that 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 a stereotype for each constraint type, as done in [14].

Next subsections define a set of patterns for the generation of the process constraints corresponding to the different typical 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.

7.1. *Sequences of activities*

A sequence flow between two activities (Fig. 6) 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.

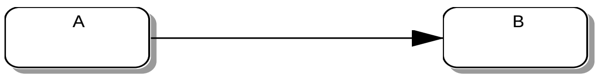


Fig. 6. Sequence flow

The first constraint (*seq*$_1$ 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 seq$_1$*: 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 6.

The other two required constraints are:

- A constraint *seq_2* over the second activity to prevent the creation of two different *B* instances related with the same *A* activity instance

  context B inv seq_2: 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 seq_3: status='completed' implies self.activity-> select(a| a.oclIsTypeOf(A) and a.status='completed')->forAll(a|a.next->exists(b| b.oclIsTypeOf(B)) and a.next->size()=1)

7.2. *Split gateways*

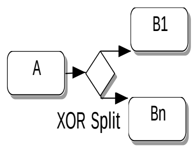
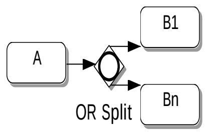
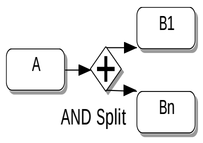
A split gateway is a location within a workflow where the sequence flow can take two or more alternative paths. The different split gateways differ on the number of possible paths that can be taken during the execution of the workflow. For *XOR-split* gateways only a single path can be selected. In *OR-splits*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

For each kind of split gateway, Table 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 1. Constraints for split gateways

Split gateway	**Process constraints**

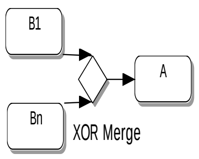
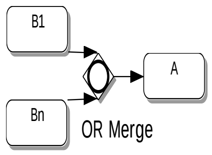
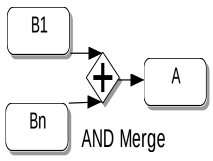
	• Only one of the $B_1..B_n$ activities may be started *context A inv: next->select(a| a.oclIsTypeOf(B_1) or … or a.oclIsTypeOf(B_n))->size()<=1* • If *A* is completed, at least one of the $B_1..B_n$ activities must be created before ending the case *context Case inv: status='completed' implies activities-> select(a|a.oclIsTypeOf(A) and a.status='completed')-> forAll (a|a.next->exists(b|b.oclIsTypeOf(B_1) or..or b.oclIsTypeOf(B_n)))*
	• Since several $B_1..B_n$ activities may be started, we just need to verify that if *A* is completed, at least one of the $B_1..B_n$ activities is created before ending the case (like in the XOR split above)
	• If *A* is completed all $B_1..B_n$ activities must be eventually started *context Case inv:status='completed' implies activity->select(a| a.oclIsTypeOf(A) and a.status='completed')->forAll(a| a.next->exists(b| b.oclIsTypeOf(B_1)) and … and a.next->exists(b|b.oclIsTypeOf(B_n)))*

7.3. *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 Ref. [42].

Table 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 the 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 2. Constraints for merge gateways

Merge gateway	**Process constraints**
	• All *A* activity instances have as a previous activity instance a completed activity instance of type B_1 or … or B_n *context A inv: previous->size()=1 and previous->exists(b| (b.oclIsTypeOf(B_1) or … or b.oclIsTypeOf(B_n)) and b.status='completed')* • Each $B_1..B_n$ activity instance is followed by an *A* activity *context Case inv: status='completed' implies activity->select(b| b.oclIsTypeOf(B_1) or … or b.oclIsTypeOf(B_n))-> forAll(b|b.next->exists(a| a.oclIsTypeOf(A)))*
	• An *A* activity instance must wait for at least an incoming flow *context A inv: previous->select(b| (b.oclIsTypeOf(B_1) or … or b.oclIsTypeOf(B_n)) and b.status='completed')->size()>=1*
	• An activity instance of type *A* must wait for a set of activities $B_1..B_n$ to be completed *context A inv: previous->exists(b| b.oclIsTypeOf(B_1) and b.status='completed') and … and previous->exists(b| b.oclIsTypeOf(B_n) and b.status='completed')* • Each set of completed $B_1..B_n$ activity instances must be related with an *A* activity instance. *context Case inv: status='completed' implies not (activity->exists(b|b.oclIsTypeOf(B_1) and b.status='completed'*

	and not b.next->exists(a| a.oclIsTypeOf(A)) and … and activity->exists(b| b.oclIsTypeOf(B_n) and b.status=‘completed’ and not b.next->exists(a| a.oclIsTypeOf(A)))

7.4. *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. 7 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* [29].

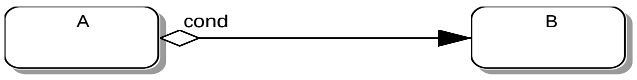


Fig. 7. A conditional sequence flow

7.5. *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. 8 shows an example of each loop type.

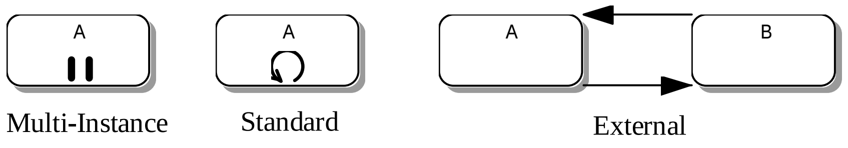


Fig. 8. Loop examples

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 outcoming 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).

7.6. *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. 9). A workflow schema may contain several start, intermediate, and end events.

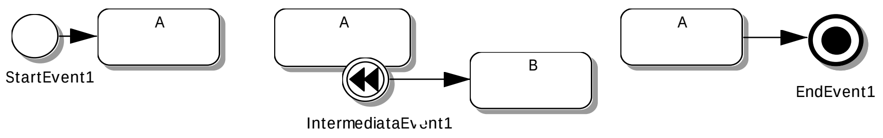


Fig. 9. Examples of events

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. Conversely, 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 previous 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.oclIsTypeOf(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. 9) 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 in Fig. 9:

- *context Event inv: eventType.name='IntermediateEvent1' and case.status='completed' implies case.activity-> exists(a|a.oclIsTypeOf(B))*
- *context Case inv: activity-> exists(a| a.oclIsTypeOf(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 the XOR-Merge gateway.

7.7. *Applying the translation patterns*

As an example, Table 3 summarizes the process constraints resulting from applying the translation patterns over the workflow running example (Fig. 4 and Fig. 5).

For the sake of brevity, in this section constraints are described in an informal and compact way. The complete set of constraints and their OCL specification is exemplified in Table 4 only for the *Provide Quotation* activity. The translation of all the other constraints is provided in the Appendix B.

The *Provide Quotation* activity involves a set of constraints due to the incoming XOR-merge from *Ask Quotation* and *Change Quotation* activities and a set of constraints due to the subsequent *XOR* split with *Submit Order* and *Change Quotation*.

Table 3. Process constraints for the workflow running example

Activity	**Constraints**
Ask Quotation	• A new *Ask Quotation* activity must be created every time a start event occurs.
Provide Quotation	• A quotation cannot be provided until an *Ask Quotation* or a *Change Quotation* finishes. A single new *Provide Quotation* instance must exist for each completed *Ask Quotation* or *Change Quotation* activity. • After providing a quotation we can either ask for a change in the quotation or submit the order, but not both (at least one of them must be executed).
Change Quotation	• The previous *Provide Quotation* activity 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).
Submit Order	• The previous *Provide Quotation* activity must be completed. Besides, only a single *Submit Order* instance must be created for the same *Provided Quotation* instance. • After submitting an order, a *Choose Shipment* or a *Standard Shipment* activities must be executed (but not both).
Standard Shipment	• The previous *Submit Order* activity must be completed. Besides, only a single *Standard Shipment* instance must be created for the

	same *Submit Order* instance. • Once the standard shipment is completed, a new *Ship Order* activity must be created.
Choose Shipment	• The previous *Submit Order* activity must be completed. Besides, only a single *Choose Shipment* instance must be created for the same *Submit Order* instance. • After choosing the shipment, both the *Arrange Transport* and *Process Order Line* activities must be executed.
Arrange Transport	• The preceding *Choose Shipment* activity instance must be completed. Besides, a single *Arrange Transport* activity instance must be executed for each *Choose Shipment* activity instance.
Process OrderLine	• The preceding *Choose Shipment* activity must be completed. • The system must exactly execute as many *Process OrderLine* activity instances as the number of order (quotation) lines for the related order.
Empty Activity1	• The new *Empty activity* instance can be created (as completed) when the transport has been arranged and all order lines have been processed. • Then, a new *Ship Order* instance must be executed before ending the case.
Ship Order	• Once a *Standard Shipment* xor an *Empty Activity1* instance has been completed, the order can be shipped. • For each order shipped, an invoice must be sent and the reception of the goods must be acknowledged by the customer.
Send Invoice	• The preceding *Ship Order* activity instance must be completed. Besides, a single *Send Invoice* activity instance must be executed for each *Ship Order* activity instance.
Receive Goods	• The preceding *Ship Order* activity instance must be completed. Besides, a single *Receive Goods* activity instance must be executed for each *Ship Order* activity instance.

Pay Goods	• An order cannot be paid until the invoice has been send and the good have been received. When both previous activities have been done, a single *pay goods* activity shall be created in response.

Table 4. Constraint definitions for the *Provide Quotation* activity

Constraints due to incoming XOR-merge	The preceding activity must be of type *Ask Quotation* or *Change Quotation* and must be completed
	context ProvideQuotation inv: previous->size()=1 and previous->exists(a| (a.oclIsTypeOf(AskQuotation) or a.oclIsTypeOf(ChangeQuotation)) and a.status=‘completed’)
	No two instances may be related with the same *Ask Quotation* or *Change Quotation* instance
	context ProvideQuotation inv: ProvideQuotation.allInstances()-> isUnique(previous)
	A *Provide Quotation* instance follows each completed *Ask Quotation* or *ChangeQuotation* activity
	context Case inv: status=‘completed’ implies activity->select(b| b.oclIsTypeOf(AskQuotation) or … or b.oclIsTypeOf(ChangeQuotation))-> forAll(b|b.next->exists(a| a.oclIsTypeOf(ProvideQuotation)))
Constraints due to the outgoing *XOR- split*	The next activity must be either another *Change Quotation* instance or a *Submit Order* instance, but not both
	context ProvideQuotation inv: next->select (a| a.oclIsTypeOf(ChangeQuotation) or a.oclIsTypeOf(SubmitOrder))->size()<=1
	If the *Provide Quotation* instance is completed, a *Change Quotation* or a *Submit Order* must necessarily be created before ending the case.
	context Case inv: status=‘completed’ implies activity->select(a|a.oclIsTypeOf(ProvideQuotation) and a.status=‘completed’)-> forAll (a| a.next-> exists(b| b.oclIsTypeOf(ChangeQuotation) or b.oclIsTypeOf(SubmitOrder)))
	Only *Change Quotation* activity instances or *Submit Order* instances may follow a *Provide Quotation* instance
	context ProvideQuotation inv: next->forAll(b| b.oclIsTypeOf(ChangeQuotation) or b.oclIsTypeOf(SubmitOrder)

8. Implementation of the Workflow-Extended Conceptual schema

Once the workflow-extended schema is available, we may automatically generate an implementation of the system that ensures a consistent behavior of all enterprise applications with respect to the business process specification.

Since a workflow-extended conceptual schema is a completely standard UML model (i.e., no new modeling primitives have been created to express the extension of the original model with the required workflow information) any method or tool able to provide an automatic model-to-code generation from UML models to a final technology platform *P* can also cope with the automatic generation of our workflow-extended schema in the same platform *P*, using general-purpose MDD techniques and frameworks.

For instance, a tool able to generate a database schema from an UML/OCL model can follow exactly the same procedure to generate a database implementation for our extended schema that guarantees the satisfaction of all workflow constraints. As usual, classes (including also the classes in the workflow subschema) will be transformed into tables, while OCL constraints (either domain or workflow constraints) will be expressed as triggers (this is not the only option, see [6] for a discussion of the different mechanisms to implement OCL constraints in databases). Similarly, a tool able to generate Java schemas from UML/OCL models could be directly used to generate a Java-based implementation of the workflow-extended schema. In this case, classes will be expressed as Java classes while constraints could be implemented as method preconditions that prevent the execution of the method if the system is not in the right workflow state.

As an example, Fig. 10 shows a possible (i) database implementation and (ii) Java-based implementation for two sequential activities *A* and *B* (Fig. 6), performed by the same user. In the former, the constraint is implemented as a trigger over the table *AtoB* representing the *Precedes* relationship type (see Fig. 5) between both activities (*AtoB* table has only two columns, *a_id* and *b_id* and records the information about which *A* activities precede each *B* activity; this is the typical database implementation for many-to-many associations in conceptual schemas). In the latter, the constraint is verified as part of the method *AssignPreviousActivity* redefined in the *B* class (corresponding to the B activity in the workflow-extended model). In both situations, when the user tries to create a new *B* activity and the previous *A* activity is not completed, an exception is raised since the *B* activity cannot start yet. The tables, triggers and/or Java classes and method bodies to implement the workflow-extended model translation (including the OCL constraints) can be automatically generated using current code-generation tools such as [12], [15], and [23] among others.

```
create trigger AtoBSeqConstraint
before insert on AtoB
for each row
Declare v_Status Varchar(10);
        EInvalidActivity Exception;
Begin
   SELECT status into v_Status
   FROM A a
    WHERE a.id = :new.a_id;
    If (v_Status<>'completed')
       then raise EInvalidActivity; end if;
End;
```

(i)

```
class B
{
  . . .
  void AssignPreviousActivity(A a)
  throws Exception
  {
     if (! a.status.equals("completed"))
          throw new Exception("Invalid Activity");
     else previous.add(a);
  }
}
```

(ii)

Fig. 10. Examples of a sequence constraint implemented in particular technologies

Note that the previous strategies to implement the sequence constraint between *A* and *B* activities are efficient ones since the constraint is only checked when linking a *B* activity instance with an *A* activity instance, regardless how many activities are part of the workflow (and the checking just compares that exact pair of instances, instead of checking all existing *A* and *B* instances). We can benefit from the fact that in our workflow-extended conceptual schema the process constraints are expressed in OCL and rely on existing methods for analyzing OCL expressions (as in [11]) to automatically compute the information about *when* and *how* checking the constraints in order to get an efficient implementation for all process constraints.

For Web applications, an interesting alternative is to fully exploit MDD approaches, such as [12] [37]: an initial hypertext model can be derived from the workflow-extended conceptual schema so that the hypertext structure enforces some of the process constraints among activities assigned to the same user[a] (or group of users) by means of driving the user navigation through the Web site. This can be done by designing in the proper way the set of pages and links that can be browsed. For instance, Fig. 11 shows a hypertext model that from the home page forces the user to go through the Web pages implementing *A* before starting *B*. The hypertext model is defined in WebML[12], a conceptual language for the specification of Web applications, already extended with workflow-specific primitives [4]. The operation units *StartActivity* and *EndActivity* are in charge of recording the information about the activities' progress in the corresponding entity types of the conceptual schema. More complicated constraints appearing in the workflow-extended model can be enforced by means of appropriate branching and task assignment primitives.

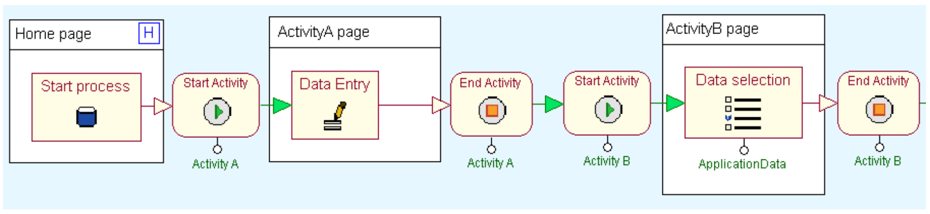


Fig. 11. Example of a sequence constraint implemented within the hypertext model of a Web application

Usually, designers will be able to choose among different strategies/platforms when implementing the workflow-extended conceptual schema. For instance, assuming a typical three tier (or n-tier) architecture, designers can decide whether to check the process constraints in the presentation layer (for example, as shown in Fig. 11), in the business layer (as in Fig. 10 ii) or in the persistence layer (as in Fig. 10 i). Each alternative may imply a slightly different behavior of the system at run-time in terms of its consistency, user experience, flexibility, reliability, and so on. For instance, a database-based implementation represents a safer alternative in terms of the data consistency (regardless how users interact with the system to update the data, the data

[a] Process constraints involving activities belonging to different users must be enforced using one of the previous techniques, they cannot be controlled at the hypertext level

will be always consistent). Instead, enforcing the constraints at the hypertext level provides a better user experience since it reduces the probability that the user actions end up in an error due to a wrong activity selection.

9. Tool Support

To show the viability of the approach, we describe the tool framework we used for realizing the whole development process presented in Fig. 1. Our framework tries to reuse as many existing tools as possible. We directly developed only the pieces that were missing for covering all the design phases. Once the designer provides the initial models, the rest of the process is performed automatically. Fig. 12 shows the tools we used for each step of the design.

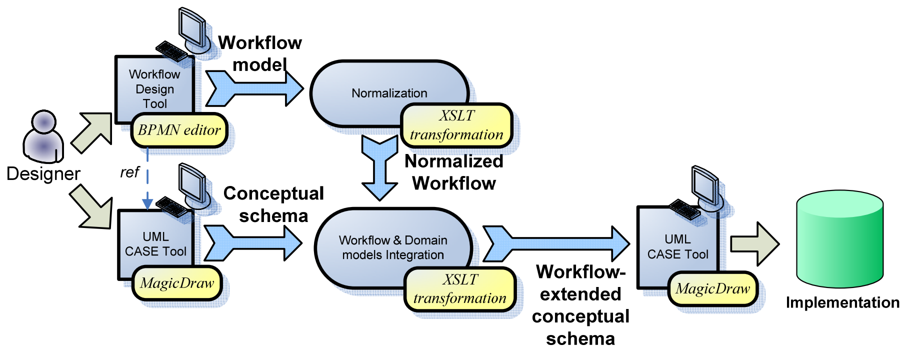


Fig. 12. Tools used for the MDD generation of Workflow-extended conceptual schemas.

The design of the conceptual schema has been done using *MagicDraw* [28] that exports it as an XMI file [31].

For workflow design and transformation, we have developed a visual editor prototype [5] that supports the design of BPMN diagrams and their automatic model transformations. This BPMN editor has been implemented as an Eclipse plugin and it is flexible and extensible. It covers the whole BPMN notation (including subprocesses) and can manage user-defined properties of objects and new transformations of the workflow models. The workflow schema is stored as an XML document according to an internal format, but proper transformations are available for importing and exporting in standard notations (e.g., to BPDM, XPDL, BPEL, and so on). The tool includes the normalization transformation, implemented as an XSLT transformation over the workflow XML representation.

Given the XML representation of the normalized workflow model and the XMI representation of the initial conceptual schema, our main transformation generates a new XMI file containing the workflow-extended model and the process constraints, according to the guidelines presented in this paper. This XMI file can be imported back and used within the *MagicDraw* tool.

10. Related Work

With respect to traditional approaches to workflow management, implemented in a plethora of commercial WFMSs, our work takes a radically different point of view, by focusing on the business process modeling and on its transformation to a software engineering specification that integrates the domain information and that can be refined and exploited by automatic code generation tools.

This approach allows for more control and personalization of the system implementation and presents a number of additional benefits as commented in Section 2. As a downside, some aspects such as integration with legacy systems, monitoring and supervision, fault management and so forth, if needed, must be provided by the application that embeds the business specification itself, instead of relying on a WFMS for providing them.

In the software engineering field, research on business process has mainly addressed the correctness of the design of the workflow model (see [18] as a representative example). Other works address the direct implementation of business process models in specific final technology platforms: for instance, [3] proposes an implementation of process constraints over relational database structures, by exploiting event-condition-action rules; [9] implements workflow models using Web technologies by mapping the workflow specification to a DSL for Web design called WebML; and [25] exploits BPEL4WS for implementing them. All these approaches are hardly reusable when trying to implement our workflow schema in different technologies or when we want to migrate our current implementation to an alternative platform. Instead, since our method works at a platform-independent level, we are able to generate an implementation of the workflow-extended method in any final technology platform using current model-driven development (MDD) approaches, as seen in Section 8. Integrating the workflow and the domain information in a single schema also allows us treating both dimensions in a homogeneous and consistent way (for instance, this enables the possibility of defining more complex business rules mixing domain and workflow information). This is not contemplated in the previous approaches.

Up to now, integration of workflows and MDD approaches has only been explored from a general framework perspective. For instance, [20] proposes to transform the workflow model to a DSL specification. However, it only provides some general guidelines for the transformation and a comprehensive framework specifying the different components that lead from the design-time specification to the runtime execution of the workflow model. However, no details are provided on the transformation rules that map a workflow model to a specific DSL. [38] proposes an approach for configuring generic process models depending on the domain information provided by the stakeholders by mean of filling questionnaires developed ad-hoc for that specific process. Questionnaires are created from the information in the initial domain model defined by the designers. Their goal is to generate, as a result, an adapted and individualized business process but not to integrate in a single conceptual schema both the process and domain information.

Some proposals (as in [41], [17], or [19]) tried to extend and adapt the UML notation for workflow modeling purposes but they did not address the unification of the business process and the conceptual schema's views of the system. 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 the platform-independent conceptual schema.

Moreover, ours is also the first translation of a workflow model into a set of equivalent OCL declarative constraints. An explicit definition of all the workflow constraints induced by the different workflow constructs is necessary regardless how these constraints are to be enforced and managed in the final workflow implementation. Very few examples of translations from process constraints to other declarative languages exist (e.g., see [8] for a translation to LTL temporal logics). In literature, OCL has only been used in relation with workflow models as an auxiliary tool for a better specification of the business process. For instance, in [16] OCL is used 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 [24] to check for well-formedness in the design of process models, in [39] to manually specify business models with UML, and in [25] to specify the contracts for the transformation of activity diagrams into BPEL4WS.

11. Conclusions

In this paper we presented an automatic approach to integrate the semantics of business process specifications within conceptual schemas. The main advantage of using conceptual schemas to handle the workflow information is that we can develop workflow-based applications without requiring the use of a specific workflow management subsystem.

Once the designer has specified both the workflow and the conceptual schemas, we build an integrated workflow-extended conceptual schema by adding to the conceptual schema *(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 conceptual schema permits a homogeneous treatment of both dimensions of the workflow-based application.

The workflow-extended conceptual schema is a completely standard UML model. This provides additional benefits. For instance, we can apply the usual model-driven development methods over our extended model to generate its automatic implementation in any technology platform. As long as these methods are able to deal with UML/OCL models, they will be able to directly manage our workflow-extended schema. In the same way, we could reuse verification and validation tools for UML models and apply them to check our extended schema.

As a further work, we would like to explore the possibility of using our extended schema as a bridge to facilitate reverse-engineering of existing applications into their original workflow models and to ease keeping them aligned. We also 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 process constraints according to the case state at that specific time) to provide a better user-experience when executing the workflow-based applications. Instead of letting the user choose the desired activity and then check whether the activity can be started, we would directly provide the list of *secure* activities avoiding possible errors in the activity selection. Along this line, we also plan to investigate the different application layers (data layer, business logic layer, presentation layer) where the process constraints can be implemented, and define some recommendation framework for the developers (and the automatically generated code) for the best implementation strategy of constraints depending on the kind of experience the application is supposed to provide to the users.

Future investigations will also address the empirical evaluation of our approach. In particular, we would like to compare the quality of manually developed applications with respect to the ones produced with our approach. For instance, we would like to compare the percentage of workflow constraints detected and included by programmers when manually developing the applications with the coverage of workflow constraints obtained when using our approach. We are confident that a manual application development will miss many workflow constraints since a manual detection of all relevant constraints and possible inconsistencies is an error-prone activity. We also plan to evaluate the effort required to develop this kind of applications with and without our approach.

Acknowledgments

This work has been partially supported by the Italian grant FAR N. 4412/ICT, the Spanish-Italian integrated action HI2006-0208 and the Spanish Research Project TIN2008-00444.

References

1. van der Aalst, W. M. P., Kumar, A.: A reference model for team-enabled workflow management systems. Data & Knowledge Engineering 38 (2001) 335-363
2. van der Aalst, W. M. P., Weske, M., Wirtz, G.: Advanced Topics in Workflow Management: Issues, Requirements and Solutions. Journal of Integrated Design and Process Science 7 (2003) 49-77
3. 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
4. Brambilla, M.: Extending Hypertext Conceptual Models with Process-Oriented Primitives. In: Proc. 22nd Int. Conf. on Conceptual Modeling (ER'03), LNCS, 2813 (2003) 246-262
5. Brambilla, M.: Generation of WebML Web Application Models from Business Process Specification. In: Proc. Tool presentation at 6th Int. Conf. on Web Engineering (ICWE'06), (2006) 85-86
6. Brambilla, M., Cabot, J.: Constraint tuning and management for Web applications. In: Proc. 6th Int. Conf. on Web Engineering (ICWE'06), (2006) 345-352
7. Brambilla, M., Cabot, J., Comai, S.: Automatic Generation of Workflow-Extended Domain Models. In: Proc. 10th Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS'07) LNCS, 4735 (2007) 375-389
8. 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 Formulae. In: Proc. 5th Int. Conf. in Web Engineering (ICWE'05), LNCS, 3579 (2005) 557-568
9. Brambilla, M., Ceri, S., Fraternali, P., Manolescu, I.: Process Modeling in Web Applications. ACM Transactions on Software Engineering and Methodology 15 (2006) 360-409
10. Cabot, J., Raventós, R.: Conceptual Modelling Patterns for Roles. Journal on Data Semantics V (2006) 158-184

11. Cabot, J., Teniente, E.: Incremental Evaluation of OCL Constraints. In: Proc. 18th Int. Conf. on Advanced Information Systems Engineering (CAiSE'06), LNCS, 4001 (2006) 81-95

12. Ceri, S., Fraternali, P., Bongio, A., Brambilla, M., Comai, S., Matera, M.: Designing Data-Intensive Web Applications. Morgan Kaufmann (2002)

13. Combi, C., Pozzi, G.: Temporal Conceptual Modelling of Workflows. In: Proc. 22nd Int. Conference on Conceptual Modeling (ER'03), LNCS, 2813 (2003) 59-76

14. 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

15. 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'01), LNCS, 2185 (2001) 104-117

16. Domingos, D., Rito-Silva, A., Veiga, P.: Workflow Access Control from a Business Perspective. In: Proc. ICEIS, vol. 3 (2004) 18-25

17. Dumas, M., Hofstede, A. H. M. t.: UML Activity Diagrams as a Workflow Specification Language. In: Proc. 4th Int. Conf. on the Unified Modeling Language (UML'01) LNCS, 2185 (2001) 76-90

18. 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

19. Hruby, P.: Specification of Workflow Management Systems with UML. In: Proc. OOPSLA'98 Workshop on Implementation and Application of Object-oriented Workflow Management Systems, (1998)

20. Hur, W., Jung, J.-y., Kim, H., Kang, S.-H.: Model-Driven Approach to workflow execution. In: Proc. 2nd Int. Conf. on Business Process Management (BPM'04), LNCS, 2080 (2004) 261-273

21. IBM. WebSphere MQ Workflow. http://www.ibm.com/software/ts/mqseries/workflow/v332/

22. ISO/TC97/SC5/WG3: Concepts and Terminology for the Conceptual Schema and Information Base. ISO, (1982)

23. KlasseObjecten. Octopus: OCL Tool for Precise Uml Specifications. http://www.klasse.nl/octopus/index.html

24. Knapp, A., Koch, N., Zhang, G., Hassler, H.: Modeling Business Processes in Web Applications with ArgoUWE. In: Proc. 7th Int. Conf. on the Unified Modeling Language (UML'04), LNCS, 3273 (2004) 69-83

25. Koehler, J., Hauser, R., Sendall, S., Wahler, M.: Declarative techniques for model-driven business process integration. IBM Systems Journal 44 (2005) 47-65

26. List, B., Korherr, B.: An evaluation of conceptual business process modelling languages. In: Proc. 2006 ACM Symposium on Applied Computing, (2006) 1532 – 1539

27. Mendling, J., Neumann, G., Nüttgens, M.: A Comparison of XML Interchange Formats for Business Process Modelling. In: Workflow Handbook. WfMC (2005)

28. NoMagicInc. MagicDraw UML v. 10.5. http://www.magicdraw.com/

29. Olivé, A.: A method for the definition of integrity constraints in object-oriented conceptual modeling languages. Data & Knowledge Engineering 58 (2006) 243-262

30. OMG: UML 2.0 Superstructure Specification. OMG Adopted Specification (ptc/03-08-02) (2003)

31. OMG: XML Metadata Interchange (XMI) Specification v.2.0. OMG Adopted Specification (formal/03-05-02) (2003)

32. OMG: UML 2.0 OCL Specification. OMG Adopted Specification (ptc/03-10-14) (2003)

33. OMG: MOF Core Specification. OMG Available Specification (formal/06-01-01) (2006)

34. OMG: Business Process Definition Metamodel (BPDM). OMG Standard, dtc/2007-07-01 (2007)

35. OMG/BPMI: Business Process Management Notation v.1. OMG Adopted Specification (2006)

36. Oracle. Workflow 11i. http://www.oracle.com/appsnet/technology/products/docs/workflow.html

37. Pastor, O., Fons, J., Pelechano, V., Abrahão, S.: Conceptual Modelling of Web Applications: The OOWS approach. In: Web Engineering. Springer-Verlag (2006) 277-302

38. Rosa, M. L., Gottschalk, F., Dumas, M., Aalst, W. M. P. v. d.: Linking Domain Models and Process Models for Reference Model Configuration. In: Proc. Business Process Management Workshops 2008, LNCS, 4928 (2008) 417-430

39. Takemura, T., Tamai, T.: Rigorous Business Process Modeling with OCL. In: Proc. OCL Workshop in MODELS'06, (2006)
40. White, S. A.: Process Modeling Notations and Workflow Patterns. BPTrends (2004)
41. Wirtz, G., Weske, M., Giese, H.: Extending UML with Workflow Modeling Capabilities. In: Proc. 7th Int. Conf. on Cooperative Information Systems (CoopIS'00), LNCS, 1901 (2000) 30-41
42. 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 A

As we have seen in Section 6, our workflow-extended schema is the result of a trade-off between the size of the model and the complexity of the OCL expression needed to represent the process constraints. However, that is not the only feasible alternative. In this Appendix we present two different alternatives: the first one aims at minimizing the size of the workflow-extended schema, while the second one tries to reduce the complexity of the required OCL expressions. The designer may choose among the three alternatives (these two plus the one presented in the main part of the paper) when following our approach for the integration of business processes and conceptual schemas.

A.1. *A minimal workflow-extended conceptual schema*

In a "minimal" workflow extended schema (Fig. 13), to reduce the size of the model, no workflow-dependent subtypes (i.e., the *Activity* subtypes recording the information about specific activities executed during the workflow case) are created. This implies that the *Activity* entity type must be extended with an additional attribute *type* to distinguish and classify the enacted activities (before we could directly determine that by examining the specific subclass of each activity instance). In our running example, the possible values for this *type* attribute are: *AskQuotation*, *ProvideQuotation*, *SubmitOrder*, and so forth.

Moreover, with this model, all relationships between the workflow subschema and the domain subschema must be done now at the *Activity* type level, instead of linking the domain classes with their specific related activities. As a result, there must exist a different relationship type between the *Activity* type and each domain class in the model (except for domain classes not related with any activity). An additional set of integrity constraints must be defined to ensure a correct instantiation of these new relationship types. For instance, in our example, an *AskQuotation* activity is only related to a *Quotation* instance. In our original workflow extended schema this was already enforced by the model itself (*AskQuotation* was only linked to the *Quotation* type) but in this minimal model, we need to add the following constraint:

context Activity inv:

self.type="AskQuotation" implies self.product->isEmpty() and

self.order->isEmpty() and self.quotationLine->isEmpty()

to ensure that activity instances of type *AskQuotation* are only associated with quotation instances. A similar constraint must be added for each workflow activity related to business data objects.

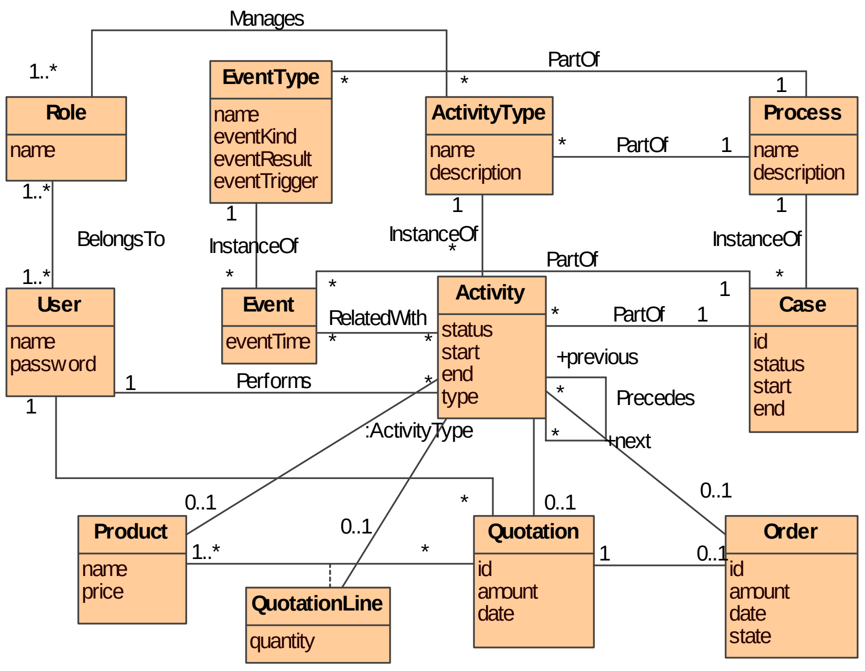


Fig. 13. A minimal workflow-extended conceptual schema

The definition of process constraints also becomes more complex. Now the *activity* type must act as context type for all the process constraints. Therefore, the first part of all constraints must be devoted to select from all activities those affected by the constraint. For instance, the first sequence constraint presented in Section 7.1:

context B inv seq$_1$: previous->size()=1 and previous->exists(a| a.oclIsTypeOf(A) and a.status='completed') is now expressed as:

> *context Activity inv seq$_1$: Activity::allInstances()->select(type="B")->forAll(b| b.previous->size()=1 and previous->exists(a| a.type="A" and a.status='completed'))*

Note that the constraints we obtain are more complex, and also the model becomes much less readable since now it is not trivial to detect all constraints affecting a particular activity type; indeed all constraints are attached to the *Activity* concept and are therefore mixed.

A.2. *A maximal workflow-extended conceptual schema*

As an opposite approach, we could prefer to sacrifice the size of model in order to get a simpler translation of the process constraints.

In this *maximal* workflow-extended conceptual schema (Fig 14), the set of workflow-dependent subtypes includes the definition of an entity type for each activity and, additionally, a different entity type for each gateway. Each gateway type is related to the activity types corresponding to the activities linked by the gateway in the workflow model. All gateway types are defined as subtypes of the *Gateway* supertype, which includes a type attribute with information on the gateway kind: AND-merge, AND-split, OR-merge, and so forth.

On the one hand, this gateway subtypes increase the size of the workflow-extended schema and complicate its management since now the system must take care of creating at run-time the appropriate instances of the gateway types whenever one of their incoming activities are completed (split gateways are automatically created as completed gateways; merge gateways are declared completed when all required incoming activities have finished).

On the other hand, process constraints can be now defined in terms of the gateways, which results in a more clear and readable definition of the constraints. That is, if an activity is affected by several gateways (for instance, an activity may be the outgoing activity of an AND-merge and the initial activity for an AND-split), the set of constraints of each gateway is attached to the corresponding gateway type instead of being mixed altogether in the activity type.

Additionally, some of the OCL constraints can be avoided because they are already enforced by the model definition itself. For instance, one of the common constraints for all split gateways among an activity *A* and a set of $B_1 \ldots B_n$ activities states that the previous activity for a B_i activity must be unique, of type *A*, and completed. The first two conditions are ensured in this maximal model due to the associations (and multiplicities) between the $B_1 \ldots B_n$ activities and the split gateway type and between the gateway and the *A* type. The condition that the *A* activity must be completed still needs to be defined as an OCL constraint, which could be expressed as simply as follows (*Split1* is assumed to be the type corresponding to the split gateway):

context Split1 inv: self.previous.completed

This situation is illustrated in the example of Fig. 14, showing the AND-Split between *ShipOrder*, *ReceiveGoods* and *SendInvoice*. Note that *ReceivedGoods* and *SendInvoice* instances must be related with an instance of the gateway, which, in turn, must be related with an instance of *ShipOrder*. This guarantees that *ReceiveGoods* and *SendInvoice* instances cannot be executed without creating first a *ShipOrder* instance.

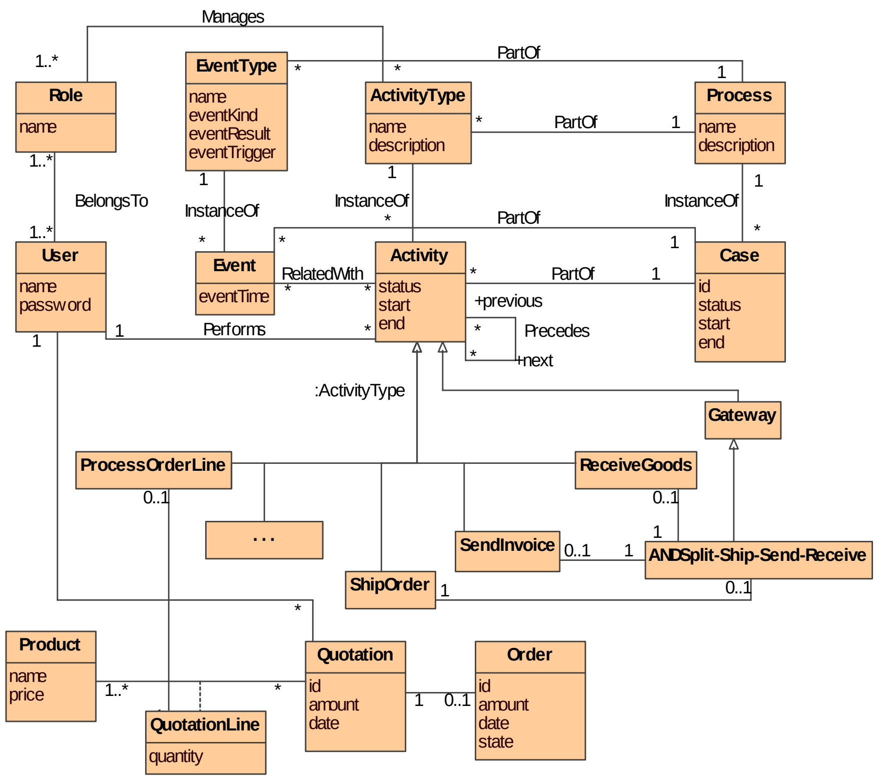


Fig. 14. A partial representation of the maximal workflow-extended conceptual schema for the workflow model of Fig. 3, showing the new entity type for one of the workflow gateways

Appendix B

The application of the translation patterns over the workflow schema of Fig. 34 produces the workflow-extended conceptual schema of Fig. 5 plus the following set of process constraints expressed in OCL.

To simplify its presentation, constraints are grouped according to the main activity they affect. For each constraint we also indicate the workflow construct generating 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 7.6.

Ask Quotation **activity**

- Constraints due to the start event
  - (i) 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.oclIsTypeOf(AskQuotation) and a.event=self)->size()=1

Provide Quotation **activity**

- Constraints due to the *XOR-Merge*
  - (i) The preceding activity must be of type *Ask Quotation* or *Change Quotation* and must be completed

    context ProvideQuotation inv: previous->size()=1 and previous->exists(a| (a.oclIsTypeOf(AskQuotation) or a.oclIsTypeOf(ChangeQuotation)) and a.status='completed')

  - (ii) No two instances may be related with the same *Ask Quotation* or *Change Quotation* instance. Note that when we iterate over the loop between

Change Quotation and *Provide Quotation* activities, new activity instances are generated in each iteration.

context ProvideQuotation inv: ProvideQuotation.allInstances()-> is-Unique(previous)

(iii) A *Provide Quotation* instance follows each completed *Ask Quotation* or *Change Quotation* activity

context Case inv: status='completed' implies activity->select(b| b.oclIsTypeOf(AskQuotation) or … or b.oclIsTypeOf(ChangeQuotation))-> forAll(b|b.next->exists(a| a.oclIsTypeOf(ProvideQuotation)))

- Constraints due to the *XOR-split*

(i) The next activity must be either another *Change Quotation* instance or a *Submit Order* instance, but not both

context ProvideQuotation inv: next->select(a| a.oclIsTypeOf(ChangeQuotation) or a.oclIsTypeOf(SubmitOrder))-> size()<=1

(ii) If the *Provide Quotation* instance is completed, a *Change Quotation* or a *Submit Order* must necessarily be created before ending the case

context Case inv: status='completed' implies activity->select(a| a.status='completed' and a.oclIsTypeOf(ProvideQuotation))-> forAll (a| a.next-> exists(b| (b.oclIsTypeOf(ChangeQuotation) or b.oclIsTypeOf(SubmitOrder)) and b.start>=a.end))

(iii) Only *Change Quotation* activity instances or *Submit Order* instances may follow a *Provide Quotation* instance

context ProvideQuotation inv: next->forAll(b| b.oclIsTypeOf(ChangeQuotation) or b.oclIsTypeOf(SubmitOrder))

***Change Quotation* activity**

- Constraints due to outgoing flow from the *Provide Quotation XOR-split*

(i) The previous activity must be of type *Provide Quotation* and must be completed

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

(ii) No two instances of *Change Quotation* may be related with the same *Provide Quotation* instance

context ChangeQuotation inv: ChangeQuotation.allInstances()-> isUnique(previous)

- Constraints due to the subsequent *XOR-merge*

(i) The next activity must be of type *ProvideQuotation*.

context Change Quotation inv: next-> forAll(a|a.oclIsTypeOf(ProvideQuotation))

***Submit Order* activity**

- Constraints due to outgoing flow from the *Provide Quotation XOR-split*

(i) 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.oclIsTypeOf(ProvideQuotation))

(ii) 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 the XOR-*split* between *ChooseShipment* and *Standard Shipment*

(i) The next activity must be either of type *ChooseShipment* or *Standard Shipment*, but not both.

context SubmitOrder inv: next->select(a| a.oclIsTypeOf(ChooseShipment) or a.oclIsTypeOf(StandardShipment))-> size()<=1

(ii) If the *Submit Order* instance is completed, a *Choose Shipment* or a *Standard Shipment* activity must be created before ending the case

context Case inv: status=‘completed’ implies activity->select(a| a.status=‘completed’ and a.oclIsTypeOf(SubmitOrder))-> forAll (a| a.next-> exists(b| (b.oclIsTypeOf(ChooseShipment) or b.oclIsTypeOf(StandardShipment)) and b.start>=a.end))

(iii) Only *ChooseShipment* or *StandardShipment* activity instances may follow a *SubmitOrder* instance

context SubmitOrder inv: next->forAll(b| b.oclIsTypeOf(ChooseShipment) or b.oclIsTypeOf(StandardShipment))

***Standard Shipment* activity**

- Constraints due to outgoing flow from the *Submit Order XOR-split*

(i) The previous activity must be of type *Submit Order* and must be completed

context StandardShipment inv: previous->size()=1 and previous->exists(a| a.status=‘completed’ and a.oclIsTypeOf(SubmitOrder))

(ii) No two instances of *Standard Shipment* may be related with the same *Submit Order* instance

context StandardShipment inv: StandardShipment.allInstances()-> isUnique(previous)

- Constraints due to the subsequent *XOR-merge*

(i) The next activity must be of type *ShipOrder*.

context StandardShipment inv: next-> forAll(a|a.oclIsTypeOf(ShipOrder))

***Choose Shipment* activity**

- Constraints due to outgoing flow from the *Submit Order XOR-split*

(i) The previous activity must be of type *Submit Order* and must be completed

context ChooseShipment inv: previous->size()=1 and previous->exists(a| a.status=‘completed’ and a.oclIsTypeOf(SubmitOrder))

(ii) No two instances of *Standard Shipment* may be related with the same *Submit Order* instance

context ChooseShipment inv: ChooseShipment.allInstances()-> isUnique(previous)

- Constraints due to the AND-split between *Arrange Transport* and *Process OrderLine*

  (i) For each *Choose Shipment* activity, the *Arrange Transport* and the *Process OrderLine* activities must be executed

  context Case inv: status='completed' implies activity->select(a| a.status='completed' and a.oclIsTypeOf(ChooseShipment))-> forAll(a|a.next->exists(b| b.oclIsTypeOf(ArrangeTransport)) and a.next->exists(b| b.oclIsTypeOf(ProcessOrderLine)))

  (ii) Only *Arrange Transport* activity instances or *Process OrderLine* instances may follow a *Choose Shipment* instance

  context ChooseShipment inv: next->forAll(b| b.oclIsTypeOf(ArrangeTransport) or b.oclIsTypeOf(ProcessOrderLine))

***Arrange Transport* activity**

- Constraints due to the outgoing flow of the *Choose Shipment* AND-split

  (i) The previous activity must be of type *Choose Shipment* and must be completed

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

  (ii) No two instances of *Arrange Transport* may be related with the same *Choose Shipment*

  context Arrange Transport inv: ArrangeTransport.allInstances()-> isUnique(previous)

- Constraints due to the subsequent *AND-merge*

  (i) The next activity must be of type *EmptyActivity1*.

context ArrangeTransport inv: next-> forAll(a|a.oclIsTypeOf(EmptyActivity1))

***Process OrderLine* activity**

- Constraints due to the outgoing flow of the *Choose Shipment AND-split*
  1. (i) The previous activity must be of type *Choose Shipment* and must be completed

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

  2. (ii) No two instances of *Process OrderLine* may be related with the same *Choose Shipment* instance

     context ProcessOrderline inv: ProcessOrderline.allInstances()-> isUnique(previous)

- Constraints due to the multi-instance loop
  1. (i) 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()->any(p|p.case=self).order.quotation.quotationLine ->size()) = 0

***EmptyActivity1* activity**

- Constraints due to the *AND-Merge*
  1. (i) We cannot start (and complete) an *Empty Activity1* instance until the *Arrange Transport* activity and all required *Process OrderLine* instances have been executed.

     context EmptyActivity1 inv: previous->exists(b| b.oclIsTypeOf(ArrangeTransport) and b.status='completed') and previ-

ous->select(b| b.oclIsTypeOf(ProcessOrderLine) and b.status='completed')-> size()=self.order.quotation.orderLines->size()

(ii) An *Empty Activity1* instance must eventually exist if the *Arrange Transport* and *Process OrderLine* activities have been issued.

context Case inv: status='completed' implies not (activity->exists(b| b.oclIsTypeOf(ArrangeTransport) and b.status='completed' and not b.next ->exists(a| a.oclIsTypeOf(EmptyActivity1))) and activity->exists(b| b.oclIsTypeOf(ProcessOrderLine) and b.status='completed' and not b.next->exists(a| a.oclIsTypeOf(EmptyActivity1))))

(iii) The previous instances of two different *Empty Activity1* instances must have an empty intersection.

context EmptyActivity1 inv: EmptyActivity1.allInstances()->forAll(s1,s2| s1<>s2 implies s1.previous->intersection(s2.previous)-> isEmpty())

- Constraints due to the subsequent *XOR-merge*

(i) The next activity must be of type *ShipOrder*.

context EmptyActivity1 inv: next-> forAll(a|a.oclIsTypeOf(ShipOrder))

***Ship Order* activity**

- Constraints due to the *XOR-Merge*

(i) The preceding activity must be of type *Standard Shipment* or *EmptyActivity1* and must be completed

context ShipOrder inv: previous->size()=1 and previous->exists(a| (a.oclIsTypeOf(StandardShipment) or a.oclIsTypeOf(EmptyActivity1)) and a.status='completed')

(ii) No two instances may be related with the same previous *Standard Shipment* or *Empty Activity1* instances.

context ShipOrder inv: ShipOrder.allInstances()-> isUnique(previous)

(iii) A *Ship Order* instance follows completed *Standard Shipment or EmptyQuotation1* activities

context Case inv: status=‘completed’ implies activity->select(b| b.oclIsTypeOf(StandardShipment) or … or b.oclIsTypeOf(EmptyActivity1)) -> forAll(b|b.next->exists(a| a.oclIsTypeOf(ShipOrder)))

- Constraints due to following AND-split

(i) For each *Ship Order* activity, the *Send invoice* and the *Receive Goods* activities must be executed

context Case inv: status='completed' implies activity->select(a| a.status='completed' and a.oclIsTypeOf(ShipOrder))-> forAll(a|a.next ->exists(b| b.oclIsTypeOf(SendInvoice)) and a.next->exists(b| b.oclIsTypeOf(ReceiveGoods)))

(ii) Only *Send Invoice* activity instances or *Receive Goods* instances may follow a *Ship Order* instance

context ShipOrder inv: next->forAll(b| b.oclIsTypeOf(SendInvoice) or b.oclIsTypeOf(ReceiveGoods))

***Send invoice* activity**

- Constraints due to the outgoing flow of the *Ship Order* AND-split

(i) The previous activity must be of type *Ship Order* and must be completed

context SendInvoice inv: previous->size()=1 and previous->exists(a| a.oclIsTypeOf(ShipOrder) and a.status=‘completed’)

(ii) No two instances of *SendInvoice* may be related with the same *Ship Order*

context SendInvoice inv:SendInvoice.allInstances()-> isUnique(previous)

- Constraints due to the subsequent *AND-merge*

(i) The next activity must be of type *PayGoods*.

context SendInvoice inv: next-> forAll(a|a.oclIsTypeOf(PayGoods))

***Receive goods* activity**

- Constraints due to the outgoing flow of the *Ship Order* AND-split
  1. The previous activity must be of type *Ship Order* and must be completed

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

  2. No two instances of ReceiveGoods may be related with the same Ship Order

     context ReceiveGoods inv:ReceiveGoods.allInstances()-> isUnique(previous)

- Constraints due to the subsequent *AND-merge*
  1. The next activity must be of type *PayGoods*.

     context ReceiveGoods inv: next-> forAll(a|a.oclIsTypeOf(PayGoods))

***Pay Goods* activity**

- Constraints due to the *AND-Merge*
  1. We cannot start a *Pay Goods* instance until the *Send Invoice* and the *ReceiveGoods* activities have been executed.

     context PayGoods inv: previous->exists(b| b.oclIsTypeOf(SendInvoice) and b.status='completed') and previous->exists(b| b.oclIsTypeOf(ReceiveGoods) and b.status='completed')

  2. A *Pay Goods* instance must eventually exist if the *Send Invoice* and the *Receive Goods* activities have been issued.

     context Case inv: status='completed' implies not (activity->exists(b| b.oclIsTypeOf(SendInvoice) and b.status='completed' and not b.next->exists(a| a.oclIsTypeOf(PayGoods))) and activity->exists(b| b.oclIsTypeOf(ReceiveGoods) and b.status='completed' and not b.next->exists(a| a.oclIsTypeOf(PayGoods))))

(iii) The previous instances of two different *Pay Good* activities must have an empty intersection.

context PayGoods inv: PayGoods.allInstances()->forAll(s1,s2| s1<>s2 implies s1.previous->intersection(s2.previous)-> isEmpty())