Modeling and Formally Checking Workflow Properties Using UML and OCL

Jens Brüning[1], Martin Gogolla[2], Peter Forbrig[1]

[1]University of Rostock, Department of Computer Science,
D-18059 Rostock, Germany
{jens.bruening, peter.forbrig}@uni-rostock.de
[2]University of Bremen, Department of Computer Science,
D-28334 Bremen, Germany
gogolla@informatik.uni-bremen.de

Abstract. In this paper, a new metamodel for workflows is described by using UML. The underlying UML class diagram is formally extended with OCL pre- and postconditions for operations and OCL invariants for system states. The metamodel allows the developer to specify processes, activities in processes and temporal relations between them. Known workflow patterns are formally captured in the metamodel and sophisticated temporal relations between activities can be expressed easily. Development of workflow models is explained as well as process instantiation and process execution on the basis of a tool realizing parts of the UML action semantics. Prototypical process execution and animation allows the designer to discover properties of the designed processes and activities in early phases of the development without the need for building a full implementation.

Keywords: Business Process Modeling, Unified Modeling Language (UML), Workflow management, IS modelling, testing and verification

1 Introduction

The importance of business process modeling languages, notations and methods for an efficient economy has been accepted in recent years without doubt. A promising approach for developing such languages is to describe them with so-called metamodels. This allows to analyze and semantically reason about the processes and their characteristics. Furthermore, this approach makes it possible to execute and simulate the designed processes in order to check for static and dynamic properties being described. We put forward a language for business processes incorporating many well-accepted process patterns on the basis of a formally specified metamodel which is expressed in Unified Modeling Language (UML). Building the language on a formal fundament gives a solid foundation for business process modeling.

UML [11] is the most popular language for modeling object-oriented software, and UML includes OCL [12] as a constraint language to specify further static and dynamic constraints. UML class diagrams together with OCL are a well-accepted means for metamodeling. As other languages, business process metamodels can be expressed by UML class diagrams as this has been demonstrated for workflows in [13, p.2-3] or [21, p.119]. Object life cycles in these business process models can be expressed by UML state diagrams (see [21, p.128] or [9, p.23]). UML class diagrams realizing the process metamodel and UML state diagrams realizing the life cycles of activities will be cornerstones of our approach.

With the business process modeling approach presented here, semantical aspects of some sophisticated workflow patterns [19] can be expressed better than in traditional modeling languages like EPCs [16], YAWL [18] or UML activity diagrams [5]. Furthermore, model inherent and explicit OCL constraints are defined in the metamodel. Such constraints can reveal invalid process descriptions. Thereby, the modeler will be guided in the development of workflows by obtaining information about violated constraints and consequently by detecting possible inconsistencies in process descriptions.

The UML tool USE (UML-based Specification Environment) [8] is applied for modeling and executing workflows. In USE, OCL invariants are permanently evaluated and violations are reported to the user. USE contains the scripting language ASSL (A Snapshot Sequence Language) and implements with ASSL large parts of the UML action semantics [11]. A constraint-oriented and user-controlled execution of workflow models is realized with ASSL.

Proceeding this way, process models can be validated. This is important for the development process of models in which the modeler has to balance between flexibility and guidance [20]. Within the validation process, the stakeholder can be involved so that a user-centric design process will be supported. The focus in this paper lies on the control flow specification of process models. The resource and data views will be embedded in future work.

The rest of this paper is structured as follows. First, the metamodel is introduced and the temporal relations between activities are expressed by UML classes, associations and OCL constraints. Second, the development of processes is shown by demonstrating how the metamodel is employed for business process model development in USE. The difference between process and process instance models will be clarified. Third, animation (or execution) of the process instances will be demonstrated. Forth, related work will be taken into account. The last section summarizes the paper and gives an outlook to future work.

2 The Metamodel for Workflow Models

This section first introduces the metamodel as an UML class diagram. All classes, associations and attributes of the metamodel are explained. Some modeling constructs for decision modeling are related to modeling elements of traditional business process modeling languages like YAWL [18].

The metamodel will focus on the control flow perspective. The modeling approach is declarative and *flexible by design* [17], which means all execution orders of the activities are allowed by default if they are not forbidden explicitly by OCL constraints. These are OCL invariants and OCL pre- and postconditions that are connected to classes and operations in the metamodel. The process modeler does not have to write OCL expressions. She can use the metamodel elements whose semantics are defined by OCL in the metamodel. The application of the metamodel for modeling workflows in USE will be presented in section 3.

2.1 Class Diagram

In Figure 1 the metamodel is shown. The central part of it is the class *Activity*. Activities have four operations that represent the interface of the activity to the user. The user can *start*, *finish*, *and skip* activities with the corresponding operations. With the operation *fail()*, the user can indicate that an activity has failed during its execution. The class *Activity* and *Iteration* have an attribute *state a*ssigned to the enumeration type *State* that indicates in which execution state the activity is currently. The rules when the state can change are specified in the life cycle presented in subsection 2.2. Further, OCL invariants can forbid the execution of an operation that will change the state of an activity, which will be subject of discussion in subsection 2.3.

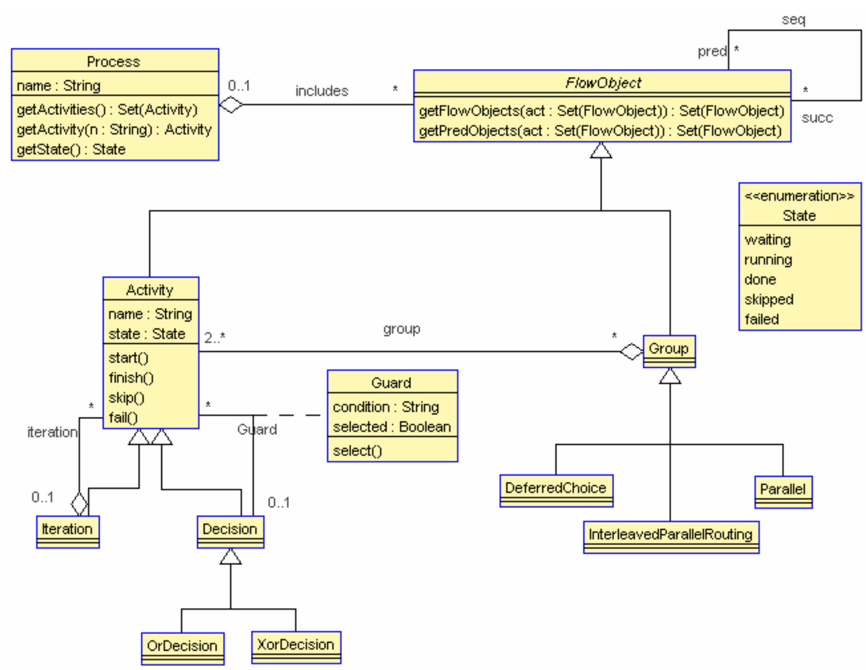

Fig. 1. The metamodel for workflow models

The activities in which decisions have to be made are modeled explicitly with the class *Decision*. By the attribute *name* of the activity the decision making task can be modeled. The conditions for selecting the right path is modeled then in the associationclass *Guard* which is connected to the subsequent activities. At runtime the user can select the desired *Guard* with the *select()* operation. Decision modeling is also part of the most process modeling languages which can be seen in Figure 2 and is intensely discussed in [2]. Similar to EPCs and YAWL, the *Decision* activity can either be an *XOR* that selects exactly one path or an *OR* that can select more than one. This fact is modeled in the metamodel with the corresponding subclasses of *Decision*.

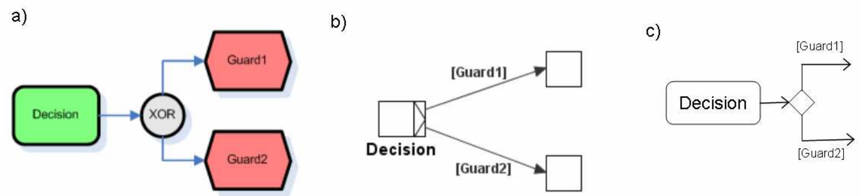

Fig. 2. Decision modeling of an exclusive choice in a) EPCs, b) YAWL and c) UML activity diagrams

Activities can be grouped together by connecting them with *Group* class. Further, the *Group* class can express temporal relations between the included activities itself. This is expressed by the three subclasses of *Group* which model three temporal relationships. *DeferredChoice* represents the Deferred Choice Pattern (Workflow Pattern 16). It is interpreted similar to the YAWL interpretation. An activity can start without an explicitly determined decision activity. When one activity out of this group is started all others of it are skipped implicitly. The other way of modeling a choice is the exclusive choice (Workflow Pattern 4) which is expressed by the class *Decision*.

The class *FlowObject* in the metamodel is a superclass to *Activity* and *Group*. The notion *FlowObject* is taken from the BPMN [14] where activities, gateways and events are combined in that term. It is an abstract class which specifies that only its subclasses can be instantiated. The association *seq* represents the sequence relationship between flow objects (Workflow Pattern 1 [19]). Thus the sequence association can be inserted between activities directly, between activities and groups or between groups itself. If a sequence relationship exists between two groups it applies to all included activities of the corresponding groups.

The Interleaved Parallel Routing Pattern (Workflow Pattern 17) is a pattern that cannot be expressed very well in traditional modeling languages like UML activity diagrams [5, p.10]. In this approach it can be expressed quite easily by connecting activities to objects of the class *InterleaveParallelRouting*. The corresponding OCL invariant expresses the temporal relation quite niftily which will be shown in subsection 2.3.

Another temporal relationship of the metamodel is not part of the Workflow Pattern catalog. It is expressed by the class *Parallel*. Activities connected by this group must be executed in parallel. Thus, if an activity of that group starts all other activities have to start as well. This does not enforce strict real time parallelism but logical parallelism. Assured real time parallelism does not appear in workflows so often but assured logical can appear frequently. An example is given in section 3 where the modeling of workflows using this metamodel will be shown.

Besides, there are several operations for practical modeling reasons that have no side effects on the object model and thus can be described by OCL expressions. The operation *getActivities()* of the class *Process* and *getFlowObjects()* of the classes *FlowObject* are those operations. They calculate the transitive closure of the activities in the process.

Having the transitive closure the process object does not have to be connected with all elements that belong to the process. Only one link between independent parts of the process have to be established by the association *includes*. This situation will be visualized in section 3 by demonstrating the development of a process model. If *getActivities()* is invoked with the respective process all its included activities will be calculated and returned.

Further, *getPredObjects()* of the class *FlowObject* calculates the transitive closure only by considering the role *pred* of the *seq* association. This is used for invariants detecting invalid process models which will be subject of discussion in subsection 3.3.

2.2 State Diagram for Modeling Life Cycles of Activities

In the previous subsection the classes *Activity* and *Iteration* have been introduced whose life cycles will be explained in this subsection. Simple activities are not allowed to be started again whereas *Iteration* activities can be started again after finishing one iteration step.

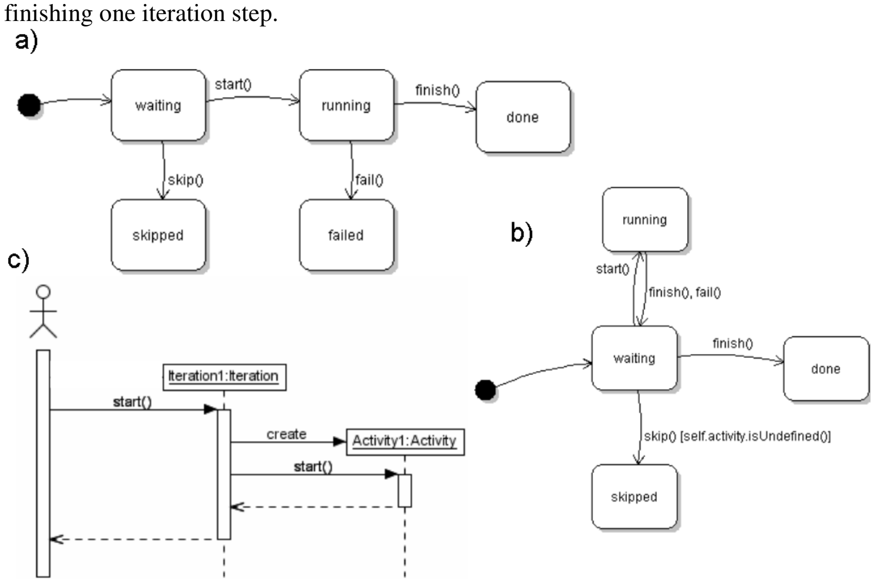

Fig. 3. UML state diagram for object life cycles of the class a) *Activity* and b) *Iteration* c) UML sequence diagram for starting an *Iteration* activity

In Figure 3 a) the life cycle of the class *Activity* is shown. An activity is initialized to the state *waiting*. Then the user has the possibility to *start* or *skip* the activity by invoking the corresponding operations. If the activity has started the user can finish the activity or mark it as *failed* by invoking the operation *fail()*.

In Figure 3 b) the life cycle of an *Iteration* activity is illustrated. It can be started whereby a new activity instance for the new iteration cycle will be created and connected with the association *iteration* to the *Iteration* activity. This behavior is pointed out in the UML sequence diagram of Figure 3 c). Afterwards the activity started inside the *Iteration* can be *finished* or *failed*. In the state *waiting* the *Iteration* can itself be finished. Afterwards it is in the state *done* and no more iteration is possible.

The compliance of the object to its life cycle modeled in the state diagram is ensured by OCL pre- and postconditions connected to the corresponding operations which will be checked by USE at runtime. The following pre- and postconditions are for the *start()* operation of the class *Acitivty*. The precondition avoids that the *start()* operation is executed on already running or finished activities. The postcondition asserts the effects of the operation to the object.

```
context Activity::start()
  pre isWaiting: state=#waiting
  post isRunning: state=#running
```

The pre- and postconditions of the remaining operations of the class *Activity* are specified analogously. According to the life cycle from the objects of *Iteration* of Figure 3 b) the pre- and postconditions have to be adjusted to the operations *finish()*, *fail()* and *skip()*.

2.3 OCL Invariants for Defining Temporal Relations

On the basis of the class diagram of subsection 2.1 and the life cycle of activities in subsection 2.2 OCL invariants can be written to define the semantics of the modeling constructs of the metamodel.

The following invariant describes the sequence relationship between activities (Workflow Pattern 1). It is specified declaratively by OCL and means: If the current activity is running, the activities connected to it with the role *pred* (which is stored in the variable *activities*) have to be in the state *done* or *skipped*. Additionally, activities that occur before the running activity can be grouped together with a *Group* object. These objects are reached over the variable *ops* by collecting all connected activities with the OCL expression *ops.activity*.

```
context Activity inv seqActivity:
    let activities:Set(Activity) = pred.oclAsType(Activity)
       ->select(a|a.isDefined)->asSet in
    let ops:Set(Group) = pred.oclAsType(Group)
       ->select(o|o.isDefined)->asSet in
    self.state = #running implies
      activities->union(ops.activity)
       ->forAll(a|a.state=#done or a.state=#skipped)
```

Figure 4 shows a valid process state concerning the invariant *seqActivity* by example. *Activity5* is allowed to be in state *running*, because its *pred* activities are either *done* or *skipped*. If the state of *Activity7* would be *running*, a constraint violation would occur, because its *pred* activity is neither *done* nor *skipped*.

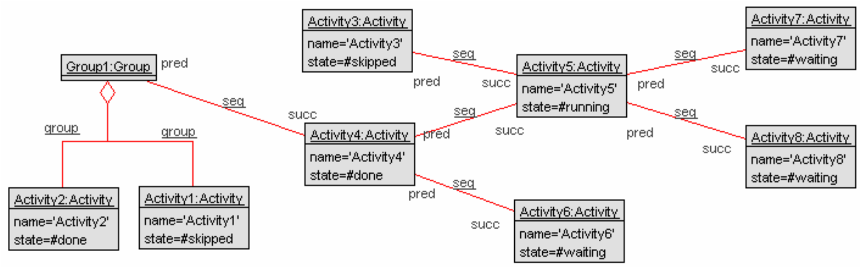

Fig. 4. Valid process state concerning the sequence relationship specified in invariant *seqActivity*

The invariant *seqActivity* is for running activities that are directly connected by the *seq* association to groups or other activities. For running activities that are first connected to a *Group* object and then indirectly to a *seq* association, a similar invariant is assigned to the Group class named *seqGroup*.

The invariants for defining the semantics of the Deferred Choice Pattern, the Interleaved Parallel Routing Pattern (from the Workflow Patterns [19]) and the assured Parallelism are assigned to the corresponding subclass of the class *Group* and are listed in the following. The Deferred Choice Pattern is expressed by the invariant of the class *DeferredChoice* which says all included activities in that group are in the state *waiting* or exactly one is *running* or *done*. In this constraint it is underspecified if the other connected activities have to be skipped or not.

```
context DeferredChoice inv allWaitingXorOneRunningOrDone:
  activity->forAll(a|a.state=#waiting) xor
  activity->select(a|a.state=#running or a.state=#done)->size=1
```

The Interleaved Parallel Routing Pattern is specified in the following invariant. This relationship is declaring that at most one activity is allowed to be running. In other process modeling languages like UML activity diagrams this pattern cannot be expressed so easily [5, p.10].

```
context InterleavedParallelRouting inv atMostOneRunning:
  self.activity->select(a|a.state=#running)->size<=1
```

The invariant of the class *Parallel* is expressing that the activities connected to that group have to be in the same state all the time. So if an activity changes the state all others have to change their states as well.

```
context Parallel inv allInTheSameState:
  self.activity->forAll(a|
          self.activity->forAll(a1|a1.state=a.state))
```

Moreover, the decision activity *XorDecision* and *OrDecision* are defined by the following invariants. After finishing the *XorDecision* activity exactly one *Guard* has to be selected. In contrast, if the *OrDecision* activity has been *done* at least one *Guard* has to be selected. The selection has to be handled by the user at runtime by calling the *select()* operation of the class *Guard*. Furthermore, the invariant connected to the associationclass *Guard* expresses that a subsequent activity can only be started if it has been selected before. By that an implicit sequence relationship is modeled between the *Decision* and its subsequent activity.

```
context XorDecision inv doneSelectedXOR:
  self.state=#done implies
    self.guard[decisionActivity]->select(g|g.selected)->size = 1

context ORDecision inv doneSelectedOR:
  self.state=#done implies
    self.guard[decisionActivity]->select(g|g.selected)->size > 0

context Guard inv onlySelectedActivitiesCanRun:
    self.activity.state=#running implies self.selected
```

Invariants can have consequences on the ASSL procedure implementation for the process animation. For example if a decision activity is finished all subsequent activities that have not been selected are skipped. Only the selected activity can be started. Thus changing the state of these activities can have side effects on other activities. Other invariants will not have consequences on the ASSL procedure implementation like the *InterleavedParallelRouting* class which only supervises the process states.

3 Example Process Modeling with USE

In this section the metamodel of the previous section is used to create workflow models. In the first subsection the process is described in natural language. In subsection 3.2 the described process is modeled in USE with the metamodel elements introduced in section 2. Resources and data are not considered in the model because they are not part of the metamodel. Only activities and control flow aspects are modeled. In subsection 3.3 invariants are discussed that detect invalid process models and support the developer in designing sensible process models. In the last subsection the differences between process models and process instance models are discussed.

3.1 The Process textually described

A process model for a peri surgical emergency process in a hospital should be developed. The process is described textually in this subsection.

The emergency process begins with the arrival of a patient by ambulance or helicopter in the emergency room of the hospital. Arrived there, the condition of the patient must be checked and a decision has to be made if she has to be operated immediately or if there is time to prepare a normal surgery. If she is in an unstable, critical situation she has to be operated immediately in the emergency room. Facilities are located there for exactly these situations. If the patient is in a stable situation she should be operated in a normal surgery in an operation room with better equipment. Surgical assistance is given by several nurses and an anesthetist and has to run in parallel to the surgery.

During the whole process the medication and medical observation of the patient runs. If medicine is given or the medication has changed, it has to be documented. The adjustment of medication and documentation of this is an iterative task which lasts until the patient leaves the hospital. The peri surgical process ends after the patient has woken up in the recovery room.

3.2 Applying USE for Modeling Processes

In Figure 5 parts of the graphical user interface of USE is illustrated. The workflow modeling elements are listed on the upper left side of the window in the *WorkflowMetaModel* tree as subnodes of the node *Classes*. These elements are the classes of the metamodel presented in section 2.

The *object diagram* view is opened in the main window of Figure 5 in which the process model is developed. Objects from the classes of the metamodel can be created. They can be connected by creating links of the associations from the metamodel.

In the process model of Figure 5 the process object *EmergencyProcess* can be seen. Not all elements of a process must be connected with the process object through links of the association *includes*. The OCL operation *getActivities()* of the class *Process* calculates the transitive closure of all modeling elements and selects the activities. Thereby, it is sufficient to connect one element of the process fragment that connects all other elements transitively to that process object as described in subsection 2.1. In the process model of Figure 5 the *EmergencyProcess* object is connected to the process elements by links of the association *includes* to the *HeliAmbu* and *AdjustMedication* object.

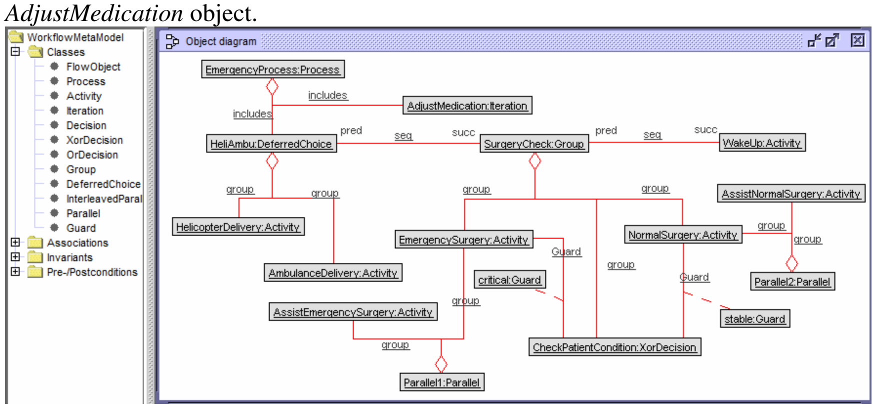

Fig. 5. Created process model: Hospital emergency process

AdjustMedication is an *Iteration* activity that is independent to all other activities in the process. The other part of the process consists of two groups of activities and the *WakeUp* activity which are connected by sequence relationships. The first group is represented by a *DeferredChoice* object. By the two activities that are part of the *DeferredChoice* group the kind of delivery of the patient is modeled. Either the patient is transported by a helicopter or by an ambulance. For these alternatives, the hospital does not have to make a decision so that the deferred choice is used in the model.

After the arrival an explicit decision is modeled with the *XorDecision* object *CheckPatientCondition*. In the connected Guards the criteria is modeled for the subsequent surgery activities *EmergencySurgery* and *NormalSurgery*. The patient is in a critical condition for the *EmergencySurgery* and in a stable condition for the *NormalSurgery*. The *Assist* activities are modeled in a parallel relationship to the surgery activities.

The last activity in the peri surgical process is *WakeUp* which stands in a sequence relationship to the activity group of *SurgeryCheck*.

3.3 OCL Invariants for supporting Development of Process Models

In the process development phase USE is supporting the modeler by forbidding insensible process models like the one presented in Figure 6. There, an excerpt of an extended model of Figure 5 can be seen. A sequence relationship between the objects *HeliAmbu* with the role *succ* and *WakeUp* with the role *pred* has been inserted. This means that the *WakeUp* activity has to be *done* before the delivery of the patient can start which is obviously wrong. USE recognizes this inconsistency by detecting a violation of the invariant *noSeqCycles*. This can be seen in the *Class invariants* view in Figure 6. The operation *getPredObjects()* is calculating the transitive closure exclusively by navigating over the *pred* role of the *seq* association (see Figure 1). The operation returns the transitive closure as a set in which it is checked if the *self* object is excluded. Otherwise a cycle is found which represents a modeling fault.

```
context FlowObject inv noSeqCycles:
  self.getPredObjects(Set{})->excludes(self)
```

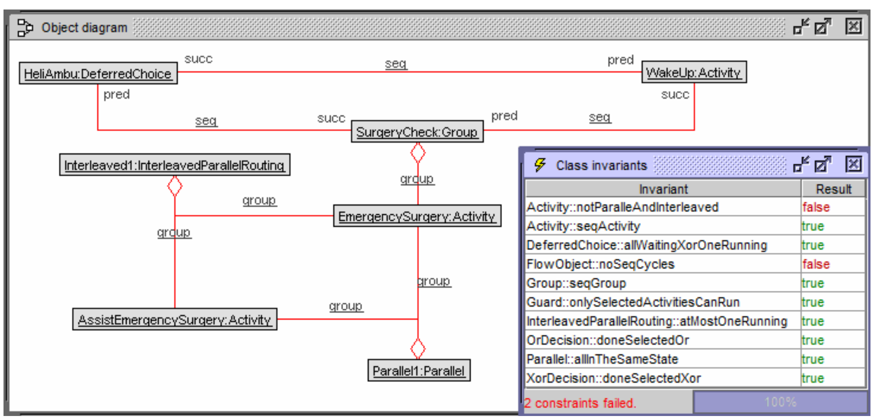

Fig. 6. Excerpt of an extended model of Figure 5 with OCL invariant violations

Another invariant is violated in the model of Figure 6. The *EmergencySurgery* and *AssistEmergencySurgery* are in a *Parallel* relationship that can be seen in Figure 5. Additionally these activities are connected by the *Interleaved1* object in Figure 6. Thus they are in the *Parallel* as well as in the *InterleavedParallelRouting* relationship which is a conflict. USE detects this conflict by calculating a violation with the invariant *notParallelAndInterleaved*. The invariant collects all parallel modeled activities connected to the *self* object in the variable *parallelAct.* Analogously all activities connected by an *InterleavedParallelRouting* Group are stored in the variable *interleavedAct*. Finally the invariant checks if those sets are disjoint by the OCL command *excludesAll*.

```
context Activity inv notParallelAndInterleaved:
  let parallelAct:Set(Activity)=self.group->select(g|
    g.oclIsTypeOf(Parallel)).activity->excluding(self)->asSet() in
  let interleavedAct:Set(Activity)=self.group
    ->select(g|g.oclIsTypeOf(InterleavedParallelRouting)).activity
    ->excluding(self)->asSet() in
  parallelAct->excludesAll(interleavedAct)
```

By the two detected modeling faults demonstrated in this subsection it is shown that USE can support the user in creating consistent workflow models.

3.4 Difference between Process and Process-Instance Models

The process model contains a *state* attribute for activities but this is not set and not interactively changed at designtime. The states will be set to *waiting* with the instantiation of the process models that are process-instance models then. With it the user can interactively change the states with the provided operations in the *Activity* class in the process animation.

The process instance model will be enriched by further information. *Timer* is a class for a singleton object that has an attribute *pit* (Point in time) which represents the current time in the process animation. Derived from this object timestamps are created during the process animation. If an activity starts or finishes, timestamps are created and assigned to the activity. By having this information the execution of the process can be logged which for example can be used for analysis purposes.

4 Process Instantiation and Animation with USE and ASSL

In this section the animation of the modeled processes will be shown. The models can be validated by process modelers and stakeholders together. Stakeholders can be interactively involved in the development phase of the process models. Appropriate balancing between flexibility and guidance [20] of the created models can be discussed by the process modeler and stakeholder on basis of this animation. Modeling faults can be detected with the animation. The validated process models can then be used for further system development. Stakeholders are involved at an early design stage of the system implementation. User centric system development can be supported by that.

In subsection 4.1 the ASSL procedures for the instantiation and execution of processes is shown. An example scenario of a process execution will be part of the subsection thereafter. In the last subsection constraint violations in process execution is presented. Potential reasons for stakeholders violating constraints are discussed.

4.1 ASSL Procedures for Process Instantiation and Animation

An ASSL procedure can be generated out of the process model that will reproduce all of the modeling elements. By that, the process model can be used as a template for the process instance models that are generated out of it. Additionally, the ASSL procedure will initialize all the activity objects in the state *waiting*.

After the ASSL instantiation procedure is derived from the process model it can be invoked for instantiating processes in an arbitrary number of times. In addition, instances of different process models can be created in the object model by invoking ASSL instantiation procedures of different process models.

In the process animation activities can be *started*, *finished*, *skipped* and *failed*. This is achieved by invoking the corresponding ASSL procedure with the object ID of the desired activity. The commands *ASSLCall* for calling another ASSL procedure, *OpEnter* for entering an operation and checking its preconditions and *OpExit* for exiting the Operation and checking its postconditions are newly implemented for ASSL in USE in the course of this work.

4.2 Successful Starting and Finishing of Activities

In this subsection a process instance execution at runtime will be shown. The interaction with the user will be logged in USE by an UML sequence diagram which can be seen in Figure 7. The ASSL command *OpEnter* represents the entering of an operation which creates a message sending arrow in the sequence diagram. The *OpExit* command generates a dashed line back to the object or user that has called the operation.

At first in the scenario the patient is transported to the hospital by helicopter. Because this activity is connected by a *DeferredChoice* object to the *Ambulance-Delivery* activity the *skip()* operation is called for that activity. The delivery finishes and the patient is checked in the activity *CheckPatientCondition*. The *Guard* “Patient in unstable condition” has been selected. After finishing the decision activity, the *NormalSurgery* activity has been skipped. By calling the *skip* procedure for activity *NormalSurgery* the object *AssistNormalSurgery* assigned by the *Parallel* operator is skipped consequently.

The sequence diagram of Figure 7 does not show the whole execution of the process. The states of the activities and assignment of the timestamps for the running and executed activities from the *Timer* singleton object is not pictured here.

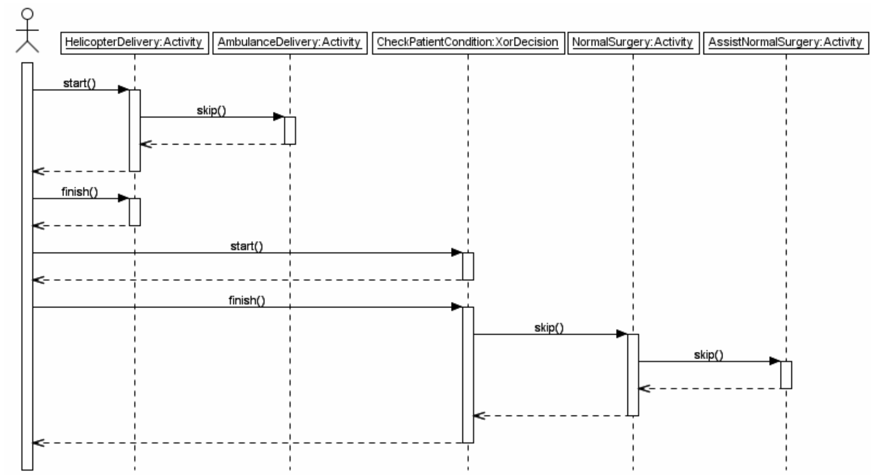

Fig. 7. Process scenario logged by an UML sequence diagram in USE

4.3 OCL Constraint Violations during the Process Animation

If an invariant is violated after an ASSL procedure execution, the ASSL commands will not take effects on the object model as long as the invariant is not deactivated. With the USE command *gen result inv* the result of the invariant evaluation is shown. In the case of Figure 8 the ASSL *start* procedure has been invoked for the *CheckPatientCondition* activity while the *HelicopterDelivery* is running.

```
use> gen start C:\M.assl start(CheckPatientCondition)
precondition `isWaiting' is true
postcondition `isRunning' is true
use> gen result inv
Note: A disabled invariant has never been checked.
An enabled and negated invariant is `valid'
if it has been evaluated to false.

   checks    valid  invalid  Invariant
        0        0        0  model-inherent multiplicities
        1        0        1  Group::seqGroup
        0        0        0  Guard::onlySelectedActivitiesCanRun
        0        0        0  InterleavedParallelRouting::atMostOneRunning
        0        0        0  OrDecision::doneSelectedOr
        0        0        0  Parallel::allInTheSameState
        0        0        0  Timer::oneTimer
        0        0        0  XorDecision::doneSelectedXor
        1        1        0  Activity::notParallelAndInterleaved
        1        1        0  Activity::seqActivity
        1        1        0  DeferredChoice::allWaitingXorOneRunning
        1        1        0  FlowObject::noSeqCycles
use>
```

Fig. 8. Constraint violation by invoking the ASSL start procedure with activity *CheckPatientCondition*

The starting of the activity *CheckPatientCondition* will violate the sequence relationship of the *HeliAmbu* object (see Figure 5). In Figure 8 this case can be seen in

USE. The appropriate invariant *seqGroup* which has been introduced in subsection 2.3 has been violated by the start procedure execution.

A reason for this scenario that has created this invariant violation could be that the stakeholder argues that the decision whether the patient has to be operated immediately or there is time to prepare a normal surgery can be made during the transportation of the patient. This would be an argument for a more flexible process model.

Another discussion initiated by the process animation is presented in the following. Within the process instance shown in Figure 9 the decision activity is finished without a selected guard. Thus the invariant *doneSelectedXor* has been violated. The scenario was continued by skipping all remaining activities. A potential argument by the stakeholder in this scenario is that the patient could be in a condition that no surgery is needed. Consequently all following activities are skipped. This argument could lead to an adjusted process model afterwards with a third *Guard* alternative assigned to the *Decision* activity.

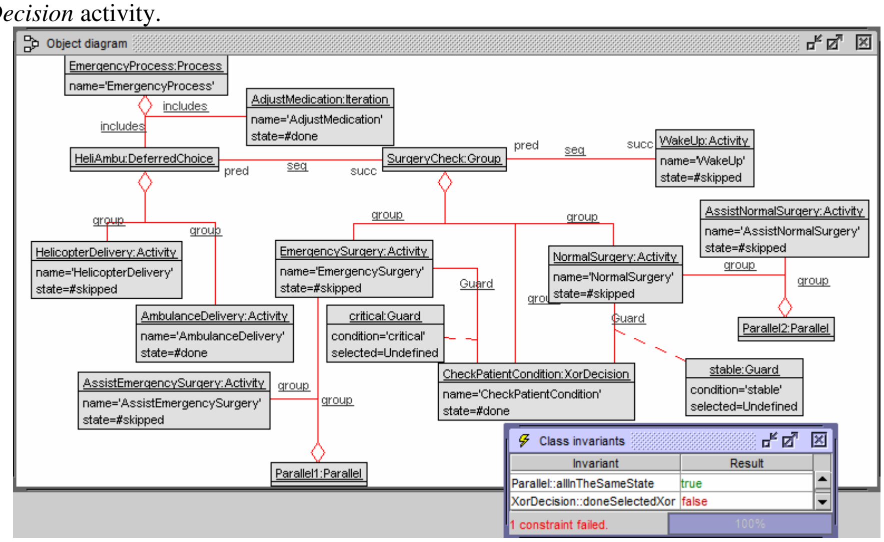

Fig. 9. Process instance after execution with constraint violation

5 Related work

Metamodeling is a widely accepted approach for defining process modeling languages. In UML the activity view is integrated with the activity diagrams which have a metamodel in the UML specification [11]. Dynamic Meta Modeling is a technique to define semantics of process languages and analyze them [6]. In [15] a metamodel approach is followed for the evaluation and comparison of different workflow management systems. A UML class representation can be used for modeling activities that are enriched with attributes and operations in the UML specification [11, p.321, p.324]. Some approaches also envision modeling workflows with UML class diagrams. In [1] domain models are enriched by process data. OCL is used in that paper as well. Previous work to this paper has used a similar approach [3], but in contrast to domain objects *User Interface Objects* have been integrated to the process models for modeling HCI and data related aspects in the workflow. In [20] a declarative, constrained-based modeling approach has been followed. The LTL language has been used instead of OCL there. For modeling workflows, the constraints are hidden behind a graphical modeling interface in the *Declare Designer* similar to the approach presented in this paper. The *Declare* models can also be executed afterwards. Validation of business processes is an important aspect that is subject to several further papers. In [10] the modeler is supported by continuous validated process models during the development at designtime. *Oryx* is a web-based process modeling tool [4] in which process models like BPMN or EPCs can be developed and validated. The validation is done by animating these models. In [7] business process models are validated by model checking techniques. Temporal logic constraints are stated and automatically checked against the process model for assuring quality. Our approach is the only one employing a formal metamodel with invariants and pre- and postconditions. This opens the possibility to check static and dynamic business process properties.

6 Conclusion

In this paper, a new way of modeling workflows is presented by using a strict metamodel approach. Semantical aspects of sophisticated temporal relationships between activities can be expressed easier than in traditional business process modeling languages like EPCs, YAWL or UML activity diagrams, for example, the Interleave Parallel Routing Pattern (Pattern 17 [19]).

Processes can be created, instantiated and animated in the tool USE. The language ASSL as part of USE has been extended for process animation purposes. Procedure calls with OCL pre- and postcondition checks can now be realized in ASSL.

Regarding future work, a process instance view within USE is currently under development in order to achieve a better interaction with the process developer. Activity calls will be encapsulated by the process instance view provided by the GUI. There, activities that do not violate any constraints will be marked as enabled. By that a constraint controlled execution of processes will be possible.

Furthermore, the metamodel will be extended with resource and data elements. This seems to be feasible especially for the data view because USE is a tool that has been developed for validating UML database models. These models can be combined with the process models introduced here. So an integrated model can be created.

Besides that, timestamps can be assigned to activity instances, and thus process instances contain data that can be used for process mining purposes. Thus a post mortem analysis of executed processes and activities can be realized by stating OCL queries which again can be evaluated in USE. Last but not least, larger case studies will give feedback on the applicability and usability of our work.

References

1. Brambilla, M., Cabot, J., Comai, S.: Automatic Generation of Workflow-Extended Domain Models, MODELS'07, Nashville, LNCS 4735, pp. 375-389 (2007)
2. Brüning, J., Forbrig, P.: Modellierung von Entscheidungen und Interpretation von Entscheidungsoperatoren in einem WfMS. EPK-2009 Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, Berlin, CEUR-WS 554 (2009)
3. Brüning, J., Wolff, A.: Declarative Models for Business Processes and UI Generation using OCL. The Pragmatics of OCL and Other Textual Specification Languages 2009, Denver, ECEASST Volume 24 (2009)
4. Decker, G., Overdick, H., Weske, M.: Oryx - An Open Modeling Platform for the BPM Community. BPM'08, Milan, LNCS 5240 (2008)
5. Dumas, M., ter Hofstede, A.: UML Activity Diagrams as a Workflow Specification Language. International Conference on the Unified Modeling Language (UML), Toronto (2001)
6. Engels, G., Soltenborn, C., Wehrheim, H.: Analysis of UML Activities Using Dynamic Meta Modeling. FMOODS'06, Oslo, LNCS 4468 (2007)
7. Förster, A., Engels, G., Schattkowsky, van der Straeten, R.: Verification of Business Process Quality Constraints Based on Visual Process Patterns. TASE'07, Shanghai (2009)
8. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming, 69:27-34 (2007)
9. Hollingsworth, D.: The Workflow Reference Model, WFMC-TC-1003, www.wfmc.org (1995)
10. Kühne, S., Kern, H., Gruhn, V., Laue, R.: Business Process Modelling with Continuous Validation. Business Process Management Workshops, BPM2008, Milan, LNBIP 17 (2008)
11. OMG Unified Modelling Language (UML) Superstructure Specification Version 2.2; http://www.omg.org/spec/UML/2.2/Superstructure/PDF/ (2009)
12. OMG Object Constraint Language (OCL) Specification Version 2.2; http://www.omg.org/spec/OCL/2.2/PDF (2010)
13. OMG Workflow Management Facility Version 1.2; http://www.omg.org/spec/WfMF/1.2/ (2000)
14. OMG Business Process Model and Notation (BPMN) Specification Version 2.0 (Beta 1); http://www.omg.org/cgi-bin/doc?dtc/09-08-14 (2009)
15. Rosemann, M., zur Muehlen, M.: Evaluation of Workflow Management Systems - a Meta Model Approach. Australian Journal of Information Systems. 6(1):103-116 (1998)
16. Scheer, A.-W.: Business Process Modeling, Springer (1999)
17. Schonenberg, M., Mans, R., Russell, N., Mulyar N., van der Aalst, W.: Towards a taxonomy of process flexibility (extended version). BPM Center Report BPM-07-11, BPMcenter.org (2007)
18. ter Hofstede, A., van der Aalst, W.: YAWL: Yet Another Workflow Language. Information Systems 30(4):245-275 (2005)
19. van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow Patterns. Distributed and Parallel Databases, 14(3):5-51 (2003)
20. van der Aalst, W., Pesic, M., Schonenberg, H.: Declarative Workflows Balancing Between Flexibility and Support. Computer Science - Research and Development, 23(2):99–113 (2009)
21. Weske, M.: Workflow Management Systems: Formal Foundation, Conceptual Design, Implementation Aspects, Postdoctoral Dissertation at University of Münster (2000)