Electronic Communications of the EASST
Volume 24 (2009)

Proceedings of the Workshop
The Pragmatics of OCL and Other Textual Specification Languages
at MoDELS 2009

Declarative Models for Business Processes and UI Generation using OCL

Jens Brüning, Andreas Wolff

16 Pages

Guest Editors: J. Cabort, J. Chimiak-Opoka, F. Jouault, M. Gogolla, A. Knapp
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Declarative Models for Business Processes and UI Generation using OCL

Jens Brüning, Andreas Wolff

University of Rostock, Department of Computer Science, Albert-Einstein-Str.21, 18059 Rostock, Germany
{Jens.Bruening, Andreas.Wolff}@uni-rostock.de

Abstract: This paper presents an approach to model business processes and associated user interfaces in a declarative way, relying on constraints. An UML-based meta-model to define processes, activities and user-interface objects is proposed. Connecting activities and user interface objects in an integrated model allows expressing interdependencies and mutual effects. Flexible execution logic for workflows and UI control flows are specified by OCL invariants. The model is constructed for the UML tool USE. Using object snapshots, USE can animate and validate business scenarios. Snapshots represent states of a process and a UI at specific times. Such animation enables business process and UI designers to discuss sensible scenarios on basis of the flexible declarative models. The intention is to create validated concrete process models in connection with UI elements that will provide a basis for the system implementation.

Keywords: Declarative Process Modeling, UI Modeling, UI Generation, UML, OCL, Constraints

1 Introduction

This paper is divided into two parts. The first one is about the declarative business process modeling approach and the connection to the User Interface Objects. These objects will provide relevant data for activities in the business process. Constraints will use them to make statements about the process logic in connection with provided data. The meta-model for the declarative approach is presented and thereafter its usage is illustrated with a case study. The process will be animated with the UML tool USE.

The second part of the paper goes into detail of the user interface aspect and discusses how to use the models of part 1 for a more detailed UI modeling and then to generate a concrete UI out of it. Problems of connecting the UI to the declarative process execution logic will be discussed here.

1.1 Motivating the declarative approach

Declarative Process Models are more flexible, because all execution paths of the activities in the process model are allowed if they are not forbidden explicitly by constraints [19, 14]. In traditional graph based process modeling languages like BPMN [1] or UML Activity Diagrams [2] sequence relationships between activities are frequently used. Those models represent concrete processes in companies or organizations. Workflow Management Systems

(WfMS) can use such models to guide the employee through the process by instantiating and interpreting them.

The sequence relationship between activities is not the standard relationship in declarative process models. By default the activities are concurrent and constraints restrict the possible activity executions. Thus, they are less adequate to guide the employees through the process because they leave too many execution possibilities up to them.

Besides, the declarative modeling approach enables the process designer to model additional temporal relations between activities that are not possible to express by the traditional graph based modeling languages. Section 2.3 will give an example for that.

Furthermore, by using the declarative approach not only temporal relations between activities can be described by constraints, also dependencies from activities to data in user interface objects can be defined. The execution logic is not only dependent on execution states of certain activities but also from data of UI-objects, which may have an impact on the process execution.

1.2 Describing the case study

For the case study in this paper we use an examination process for an undergraduate informatics course at our university. So it is about an administrational business process in an organization and not in a company. The examination process in the Abstact Datatype lecture [3] will be modeled and is as follows.

The student gets exercise sheets that he has to solve at home. This homework period includes iterations for each sheet and is running until the last exercise is done. Further on, the student has to do three written tests during the lecture. Because of that we consider a test as an iterated activity. The period of homework begins earlier and runs longer than the test period.

These two partial examinations are preconditions to write the final written examination test.

2 Concept of Declarative Process Models using OCL

First of all we describe the meta-model for declarative business processes in general. Thereafter, a concrete process is described by a case study about the lecture examination process that was textually described in Section 1.2. The business process logic is described declaratively by OCL-invariants in Section 2.3. Dependencies of activities in the business process to the data from the user interface are also described by OCL-invariants in the section thereafter. At last the case study is animated by a concrete scenario in the UML tool USE. That way the declarative process model can be tested. Also suitable diagrams to analyze the scenarios are shown in Section 2.5.

2.1 Meta-model for declarative processes

In Figure 1 the process meta-model is presented. The model introduced in [4] is extended with iterations here.

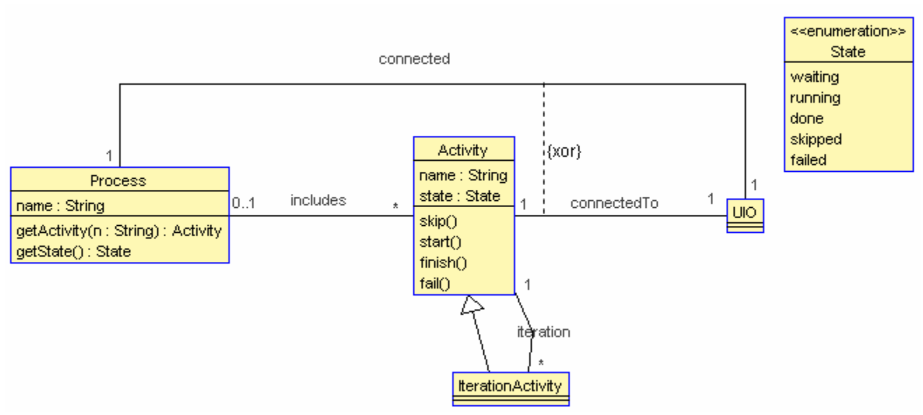


Fig. 1. The meta-model for the declarative business process modeling approach

The class *Process* has an attribute *name* and contains a set of activities which is described by the association *includes* between *Process* and *Activity*. Atomic actions and actions that occur iteratively in the process are expressed by the class *Activity*. They have a name that describe the actions and a state in which the activity is currently in. Possible states for the activities are described in the enumeration *State*. The process state is derived from the states of the included activities and is calculated by the operation *getState* in the class *Process*.

Further on, the operation *getActivity* returns the requested activity instance and is needed by the subsequent invariants presented in Section 2.3 which describe the business logic. It is coded in OCL [8] as follows and will be interpreted by the UML tool USE [5]:
getActivity(n:String):Activity = activity->any(a|a.name=n)

An *Activity* can have a set of *IterationActivity* that describes the action in every iteration cycle. The following invariant expresses the relationship between the parent activities to its iteration instances. If the parent activity is in the state *done* the iteration instances have to be *done* as well.

```
context Activity inv IterationActivities:
   self.state=#done implies self.iterationActivity->forAll(a|a.state=#done)
```

At last, there are two associations to the class *UIO* that provide the connection to the UI-Objects. Thereby, constraints can describe dependencies between business process and data in the UI-Object. Dependencies from the direction of the process to the UI-Objects will be described in Section 2.4. Constraints that describe effects from the UI to the business process will be part of Section 3. Also, details of the UI models and special interface types will be introduced there.

The lifecycle of the activities is important for the business process logic described by OCL invariants in Section 2.3. It is shown in Figure 2. The activities are initialized to the state *waiting*. After invoking the operation *start* on an activity instance, the state will be changed to *running*. The activity instance can terminate in the states *done*, *skipped* or *failed*.

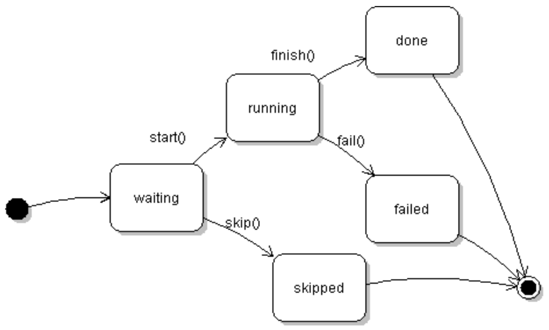


Fig. 2. The state chart models the life cycle of instances of the class Activity in the meta-model

The behavior specified in the state chart will be modeled by OCL pre- and post-conditions for all of the operations listed in the class *Activity* in the UML tool USE. For example the pre-condition of the operation *start* specifies that the corresponding activity is in the state *waiting* and the post-condition states that it has changed to *running*. In OCL this is expressed as follows and the conditions of the further operations are specified analogously.

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

2.2 Concrete process level

The concrete process level is achieved by inheritance relationships between the meta-class and the modeled subclasses which can be seen in Figure 3. The process *LectureADT* represents the concrete process in there. It is connected with 3 activities that are described by the concrete activity classes *HomeworkPeriod*, *TestPeriod* and *Exam*. The *HomeworkPeriod* and *TestPeriod* are iteration activities that include the activities *Homework* and *Test*.

Process definition invariants are specified for the subclass *LectureADT*. First of all the names of the included activities are expressed in the first invariant. Then the second invariant ensures that selected activities are of the corresponding types. Further on, the iteration activities can be connected to the period activities in an analogous way so that these invariants are omitted here.

```
context LectureADT inv LectureExamination:
   self.activity.name = Bag{'TestPeriod','HomeworkPeriod','writeExam'}

context LectureADT inv SpecialOperations:
   self.getActivity('TestPeriod').oclIsTypeOf(TestPeriod) and
   self.getActivity('HomeworkPeriod').oclIsTypeOf(HomeworkPeriod) and
   self.getActivity('writeExam').oclIsTypeOf(Exam)
```

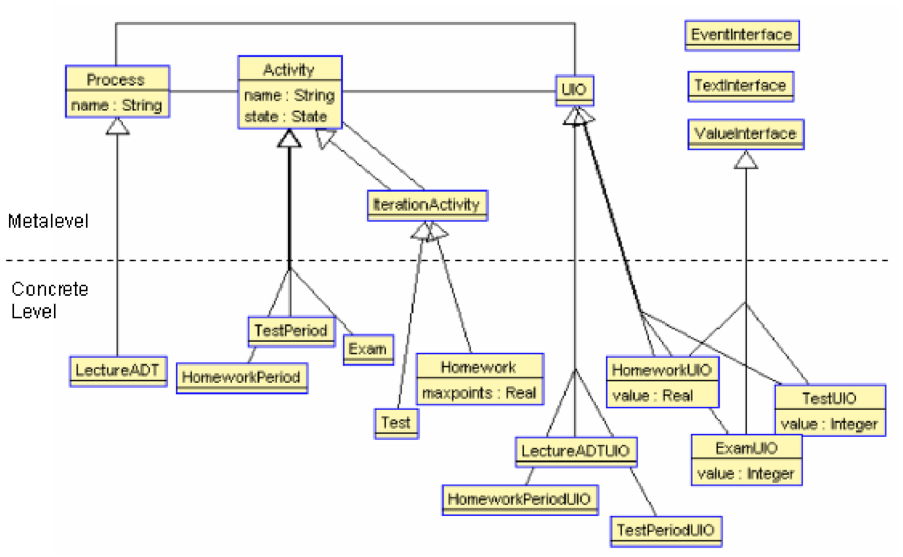


Fig. 3. The relationship between the meta level and the concrete modeling level. Concrete Process, Activities and UI-Objects are represented by different classes in the concrete level.

2.3 Business process logic described by invariants

The process logic is described by OCL invariants. There are several well known temporal relationships between activities like the sequence relationship or the iteration that can be described in that way. These relationships are often used in the typical process modeling languages. In the examination process of our case study there is a sequence relationship between *HomeworkPeriod* and *Exam*. It is described in the invariant *Homework_Exam_Sequence* which can be seen beneath. The same relationship exists between *TestPeriod* and *Exam* and can be specified in a similar way.

With our declarative process modeling approach we can describe additional temporal relationships that cannot be expressed by the common process modeling languages [4] and are not part of the workflow patterns [6]. The invariant *TestPeriod_in_ HomeworkPeriod* is an example for that. It expresses that the Test-period has to be during the Homework-period with no further temporal limitation.

```
context LectureADT inv Homework_Exam_Sequence:
  self.getActivity('writeExam').state=#running implies
    self.getActivity('HomeworkPeriod').iterationActivity->forAll(a|a.state=#done)

context LectureADT inv TestPeriod_in_HomeworkPeriod:
   self.getActivity('TestPeriod').state=#running implies
      self.getActivity('HomeworkPeriod').state=#running
```

The OCL constraints for expressing the temporal relationships are quite verbose. To make this approach more practicable, the meta-model can be extended with more subclasses derived from the class *Activity* that have particular properties expressed by predefined constraints similar to the class *IterationActivity* introduced in Section 2.1. For example subclass *SequenceActivity* can be derived from the class *Activity* and an association is connecting the activities that are subsequent in the process logic. At design time the process modeler only has to specify which activities have to be connected instead of writing the verbose constraints.

To support the process modeler while designing the process models, invariants for the declarative process models could be generated out of a graphical oriented modeling tool similar to the *DECLARE Designer* [18]. So, such a modeling tool would be wishful also for the approach presented here. More complicated relationships can be added by specifying explicit OCL constraints afterwards.

2.4 Dependencies of activities to User Interface Objects

An advantage of our declarative process modeling approach is the access to the UI elements and the data in there. We can describe specific states of the business process by referencing the states of including activities and then make constraints to the expected data from the user interface. The invariant *ExamOnlyIf_HomeworkPassed* describes that if the exam is running the tests written before have to be passed with a rate better than 50 percent. For the successful passing of the homework also 50 percent of the maximal points have to be collected. This issue is proven with the invariant *ExamOnlyIf_HomeworkPassed*.

```
context Exam inv ExamOnlyIf_TestsPassed:
   let percents:Bag(Integer) =
      self.process.
        getActivity('HomeworkPeriod').iterationActivity.uIO.oclAsType(HomeworkUIO).getValueAsInt() in
   self.state=#running implies
      percents->sum() / percents->size() > 50

context Exam inv ExamOnlyIf_HomeworkPassed:
   let homeworks:Set(Activity) = self.process.getActivity('TestPeriod').iterationActivity in
   self.state=#running implies
   (homeworks.uIO.oclAsType(HomeworkUIO).getValueAsReal()->sum() /
      homeworks->collect(a|a.oclAsType(Homework).maxpoints)->sum()) * 100  > 50
```

A further aspect for the whole process state can be expressed by referencing data from the corresponding UI-object. By overwriting the operation *getState()* of the meta-class *Process* in the concrete process *LectureADT* we have the possibility to express that the whole examination process is only done if all included activities are done and the UI-object of the exam has a grade better than 5 which means successfully passed in Germany. If it would have been a 5 the examination process would have failed. The other two possible process states are *running* and *waiting*. The operation *getState()*, to calculate the applicable state, can be seen in its following OCL-specification.

```
getState():State=
 if activity->forAll(a|a.state=#done) then
      if getActivity('writeExam').uIO.oclAsType(ExamUIO).getValueAsInt() < 5 then #done
      else #failed endif
else if activity->exists(a|a.state=#running) then #running
 else #waiting
endif endif
```

2.5 Process animation in USE

The process can be instantiated and animated in USE in connection with ASSL (A Snapshot Sequence Language) [5, 7]. USE interprets the ASSL procedures and commands take effect on the object model like creating, deleting or changing objects or links. The model of Figure 3 is specified for USE and an ASSL process instantiation procedure will create the process instance.

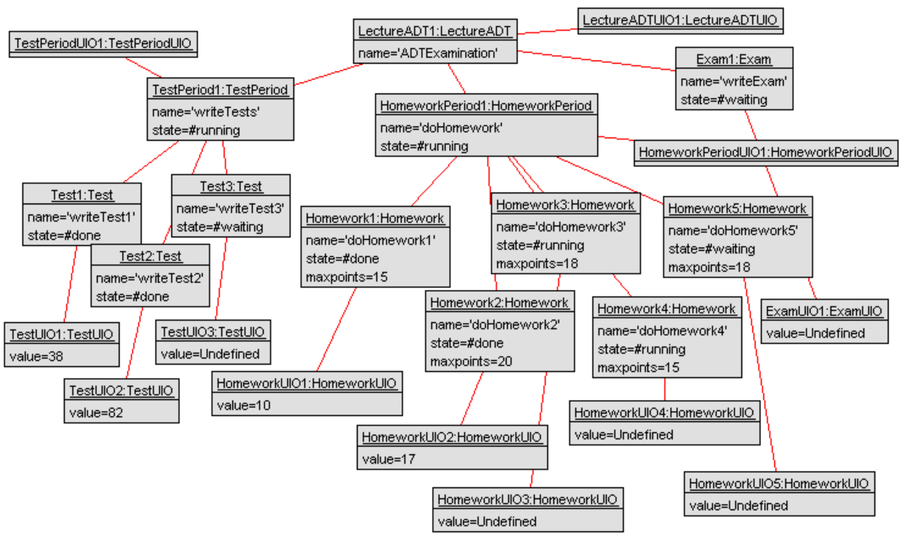


Fig. 4. A snapshot of the process instance with activities and corresponding UI objects

The ASSL commands in that procedure will result out of the process definition invariants of Section 2.2 that have already specified which activities are included in the process. For example in invariant *LectureExamination* the activity "writeExam" is defined and its type *Exam* is assigned by the invariant *SpecialOperations*. Further, the initial state *waiting* of the activities are set in the ASSL instantiation procedure and the UI-Objects will be created and linked to the corresponding activities as well. Figure 4 is showing such a process instance.

Having the process instance, activities can be started, finished, skipped or failed. These commands can also be expressed in ASSL procedures. An ASSL procedure invocation is executed only if no invariant is violated after the execution of the procedure run. Otherwise USE informs the user that no valid state is found and the ASSL commands will not take effects on the object model. The invariants are checked and monitored at runtime by USE. By that a constraint controlled execution of the process model is guaranteed.

Figure 4 is showing a snapshot of a process after instantiating it and invoking some *start()* and *finish()* operations on several activities that have not violated any constraints. The invocation sequence of the operations will be logged in the sequence diagram in USE and can be seen in Figure 5.

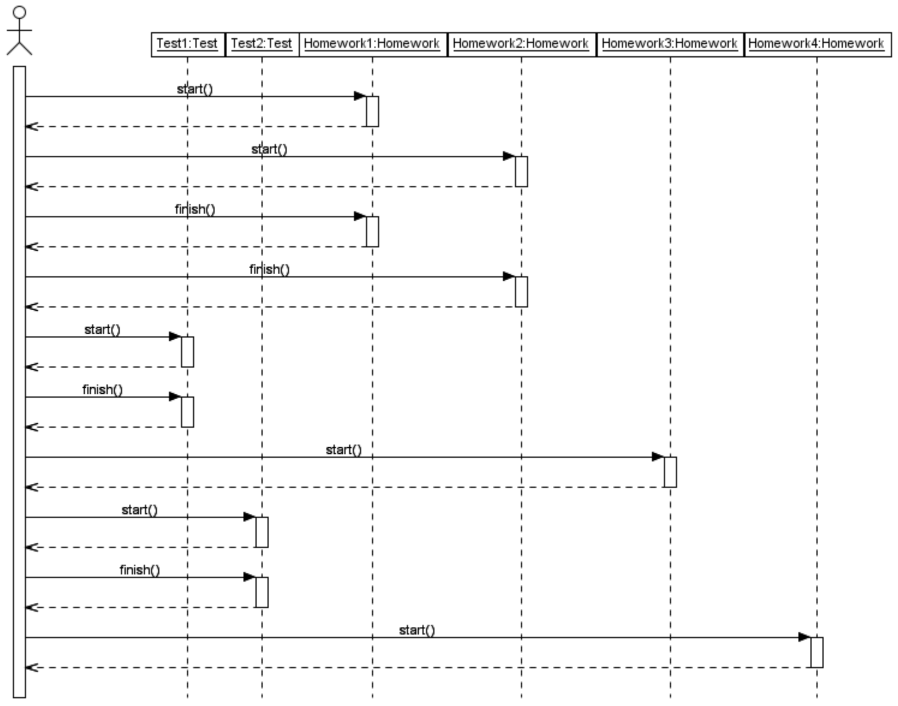


Fig. 5. A sequence diagram showing the scenario until the snapshot of Figure 4 is taken

The ASSL operations for starting and finishing the activities have to be invoked at the console in USE. For validating the process definition by animating and simulating process instances, it would be wishful to have a process view in USE where process executions can be visually animated and the ASSL procedure-invocations for starting and finishing activities are provided by a GUI.

3 Models for UI

3.1 User Interface Modeling

Modeling user interfaces (UI) is a well investigated research topic. For more than twenty years user interfaces are specified in terms of instances of meta-models. Approaches differ in level-of-detail and abstraction, domain, interface modalities and target devices. Yet, most common and among the earliest, were model-based systems for text-based interfaces and typical Window-Icon-Menu-Pointer (WIMP) style graphical user interfaces.

In model driven user interface development (MD-UID), as well as in model-driven software development, a distinction is made between platform independent (PIM) and platform specific models (PSM) [13]. A platform in MD-UID is considered to be a certain device or rather the interface source code itself. Interface source code is not necessarily source code of certain, more or less well known, programming languages. More and more interfaces are described in a markup language. A very well known example is HTML.

Closely related to model-based user interface development are any types of XML-based UI-languages. Their main advantage is an available well-defined grammar and their hierarchical structure. Because of these characteristics the definition of an MOF like meta-model is comparatively easy.

Two XML-based UI languages can be considered wide-spread: XAML, which is used throughout Microsofts .NET framework, and XUL [9] of the Mozilla Corporation, used in the FireFox web browser and other products. Other XML-UI languages were developed for research purposes and are mostly used in academic context. These include for example UsiXML, UIML and XIML.

MD-UID concepts are often based on at least three interrelated models. Those models are found in early systems like e.g. MASTERMIND [15] as well as current systems as e.g. Zhao's [17]. One of these models is a task model which describes the activity a user wants to accomplish in terms of goals, sub-goals and temporal relationships. A presentation model defines layout, interaction objects and control flow. And the domain model which describes the application domain, i.e. its objects or other environmental constraints. Using declarative modeling to describe user interfaces is also a well-known concept in MD-UID, FUSE [16] of 1995 was an early system dedicated to formal definition of user interfaces.

Those models are found in our approach as well, though not as explicit models. The invariants of Section 2.3 define temporal relations and goal decomposition very similar to a task model. We consider the concrete process model as our domain model. Obviously, this is not the best solution one can think of. We had some discussion whether to introduce a specific data model, but eventually decided not to. The major reason was brevity and that domain modeling is not the core of the approach presented here. The remainder of this paper describes the presentation model of our approach.

In the meta-model for the business process modeling approach we use a very concise user interface model. It basically consists of the abstract class *UIO* and the interfaces *ValueInterface*, *TextInterface* and *EventInterface*. The name *UIO* is short for user interface object which is a generic expression for any element on any kind of human-computer interface.

Class *UIO* does not specify any attributes or operations. Its main purpose is to serve as super type within an inheritance hierarchy. The interfaces mentioned before are intended to define fixed and in particular *typed* methods of accessing internal state and value information of concrete user interface objects.

Every concrete user interface object has to be an instance of a subclass of *UIO* and should implement one or more of the interfaces, whichever is appropriate.

Figure 6 shows the user interface part of our workflow meta-model. We consider those four types as sufficient to specify a platform independent model of an user interface.

As platform specific model for UIs we decided to use XUL. One reason for this decision was the availability of an advanced rendering engine, which is open-source and independent of the underlying operating system. Furthermore we already have a visual editor for XUL at our disposal. And more over, XUL is also used as main UI-language in our other research.

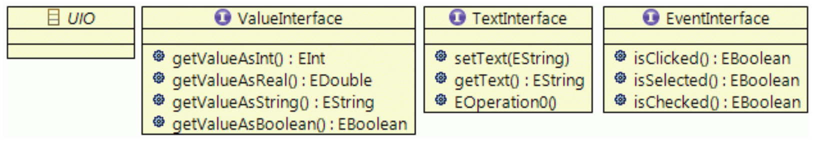


Fig. 6. Types for linking to user interfaces in meta-model

To use XUL as PSM for UIs within our approach we first had to define a MOF-based meta-model for it. So we developed an EMF Ecore [11] model for this purpose. We chose an Ecore model over a USE model because we found the support for large models in USE insufficient. This is mainly because USE does not offer packages or interfaces.

Using freely available sources it is possible to construct a XML schema definition (XSD) for XUL. Having a schema definition, a first Ecore model for XUL was easily generated. The generated model had a number of disadvantages, e.g. many numbered or anonymous types, and we found it was unnecessarily complicated. So we manually refined and restructured the model.

Figure 7 visualises the basic hierarchy within the XUL Ecore model. Many types, either representing concrete user interface elements or used to structure the model, were omitted to not waste space. To give an idea about the size: the model consists of 31 data types, 151 classes and interfaces with 443 attributes in total.

XULElement is the root element of the type hierarchy; it has a lot of attributes which are valid for every XUL tag. Every attribute is initialized with a sensible default value or remains unset if no value is required explicitly.

TemplateControl is the root type for any XUL tag that controls the template engine. *InfoElement* is the super type of every tag that is not displayed but defines actions, key bindings and such.

Any type that implements *ContainerChild* can be placed into a visual container. Most implementors are concrete *VisibleType* objects, some are visual containers themselves. *ContainerElement* is a marker class for all elements which are able or require containing sub-elements.

A *GenericContainer* object is a visual container that includes children of type *ContainerChild*. An example could be a group box, which is a bordered box that has a caption, which has some labels and buttons in it.

RadioType is a radio-button, *ButtonType* a plain button, *LabelType* a label and *TextboxType* a textbox. Those four types are the only concrete classes of Figure 7. Each such type represents a concrete user interface object.

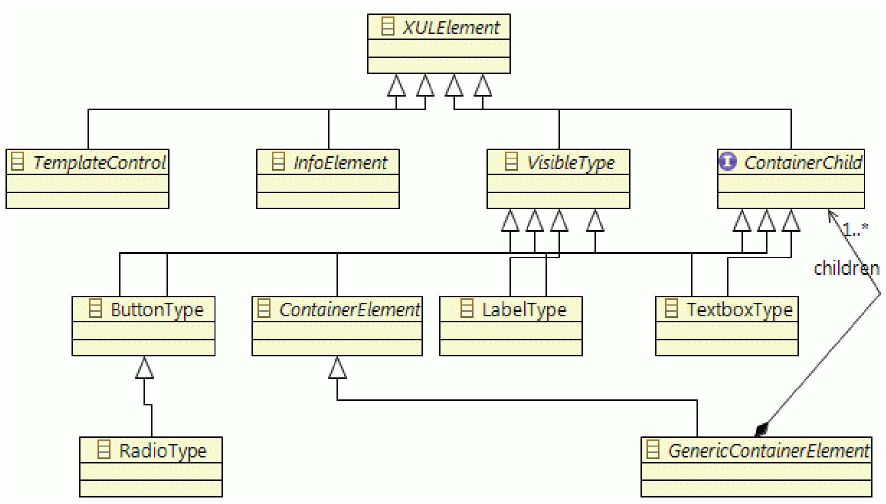


Fig. 7. Section from XUL Meta Model

The XUL meta-model we developed can be used as a descriptive model for XUL user interfaces. Beside it's usage in the context of this paper, it is the foundation for a number of tools we are currently working on in MD-UID.

3.2 Coupling Workflow Modeling and User Interfaces

Section 3.1 presented two kinds of meta-models. Now we would like to show how the combination of both can be used to connect user interface objects to a workflow.
It was mentioned that the workflow model of Section 2 includes references to platform independent user interface objects. Those are specified in a platform independent manner using the type *UIO*.

To obtain concrete user interface objects we merge the XUL meta-model and the UI model of the workflow meta-model. To achieve this multiple inheritance is used.

Every user interface object of an activity or a process object must be a member of a concrete class. Such class has to inherit from class *UIO* and from its representing XUL meta-type class. If needed, it also would inherit or implement the appropriate value type interface.

To give an example: The workflow activity *Homework* might be presented using a textbox to input the points a student achieved in this particular assignment. Therefore a user interface object is needed that provides such an input and is able to disclose the point count to any stakeholder or constraint. For this purpose the class *HomeworkUIO* is defined within our concrete workflow model. It inherits from *UIO*, from *TextboxType* and implements *ValueInterface*. The attribute uIO of type *Homework* in our concrete workflow is able to hold references to *HomeworkUIO* objects.

When generating code one would implement the method *getValueAsReal()* with an implementation like: *return parseFloat(this.value)*. The value attribute is an XUL attribute of textboxes and it is available in *HomeworkUIO* through inheritance from *TextboxType.*

Often one would like to group similar user interface objects within a visual container. As we decided to keep our UI-PIM most concise, the *UIO* does not provide any mechanism to specify parent/child relations or containment associations. This was a deliberate decision, which was made to keep the focus on the declarative workflow aspects in our meta-model.

Nevertheless containment relations can still be modeled through the platform specific model. We consider grouping as an important layout problem, but not as a crucial part of a platform independent workflow UI model.

For specifying parent/child relations we can easily use *GenericContainerElement*'s containment association *children*.

In continuation of the example for the *Homework* activity we may want to group all *HomeworkUIO* into a certain screen area. The *Activity* type *Homework* was specified as an *IterativeActivity* and its instances are repeated within a *HomeworkPeriod* object.

We would now define a class *HomeworkPeriodUIO* that inherits from *UIO* and *GroupboxType*. *GroupboxType* is a subtype of *GenericContainerElement* and also of *ContainerChild*. This means that a XUL groupbox contains child elements and also may be a child in another container. We do not implement any of the value conversion interfaces, because there is no sensible value or state to report to anybody.

In an instantiated workflow *HomeworkUIO* objects may now be placed as children of a *HomeworkPeriodUIO* object. In fact any *UIO* object of the workflow instance that implements *ContainerChild* may be placed in this group. If such freedom is unwanted, restrictions can be introduced by appropriate constraints.

We do not intend to describe the whole user interface of a workflow declaratively. But nevertheless it seems sensible to define some constraints on the PSM level of a UI. Other sample constraints might be the requirement that the *UIO* of a process is a *ContainerElement* or that every visual container needs to have children.

Of special interest in the context of this paper are constraints that take effect on the workflow. By integrating the *UIO* into the workflow meta-model we provided a mean of bidirectional control. That is we can control the UI via workflow and vice versa.

We assume that constraints from the user interface will mostly be based on the internal state of a certain *UIO*. Once again the value and state interfaces with their typed methods are used.

An example of a constraint that affects the workflow can be defined in the context of *HomeworkUIO*. We may define that the *Homework* activity which is connected to a certain *HomeworkUIO* can never be in the state of *'done'* if there is no text entered into the *UIO*'s textbox. This constraint might be defined as follows:

```
context HomeworkUIO inv AssignmentPoints:
            self.value.oclIsUndefined()  implies  self.activity.state<>#done
```

More complex constraints are of course conceivable, e.g. for a *HomeworkUIO* a constraint which is composed over all its child-*uio*'s. However, almost all constraints will probably declare implications for the state of a certain activity object.

3.3 Runtime User Interface

The UI model which we specified using the methods presented in Section 3.2 is sufficient to generate a functional user interface. Unfortunately the tool chain we use is not.

For the time being USE does not provide a feasible method of interaction with other tools. A plug-in interface is under development, but not available right now. We do need such an interface to create an adapter or proxy between our selected UI renderer and USE.

In our case the renderer of the PSM UI-model could be any instance of the gecko rendering engine of the Mozilla Corporation, as for example the FireFox web browser.

Some of the features of USE are available within the eclipse platform and EMF as well. To bridge the time gap until USE becomes fully usable to us, we attempt to validate our approach within EMF. The rest of this section describes how concrete user interface can be generated from our meta-model and what kinds of checks to the UI model are still possible.

As mentioned earlier, our target UI language is xml based. The task to convert the instantiated concrete workflow UI model into a valid XUL file is assigned to a template engine. We decided to use openArchitectureWare [10] (oAW) XPand. One reason was that Xpand's template is polymorphy-aware; also it provides functional extensions and a model validation mechanism.

Before generating any user interface code we have to specify all *UIO* types, instantiate a workflow, create the required number of *Activity* and *UIO* instances and associated them among each other.

Figure 3 shows all UIO classes specified for the workflow of our lecture example. Table 1 gives the type definitions for all five concrete UIO types after inheriting from the XUL and workflow meta-models.

Type	Supertypes
LectureADTUIO	UIO, WindowType
HomeworkPeriodUIO	UIO, GroupboxType
TestPeriodUIO	UIO, GroupboxType
HomeworkUIO	UIO, TextboxType, ValueInterface
TestUIO	UIO, TextboxType, ValueInterface

Table 1. Effective supertypes of concrete UIO types

An instance of our concrete workflow contains the appropriate amount of objects of each type. The instantiated concrete workflow is available as Ecore model and should include the concrete UIO objects. Figure 8 depicts such an example.

The next step towards the generation of a XUL file is model validation. OpenArchitectureWare includes a dedicated language, named Check, to specify validation constraints. We use it to do some consistency checks. For example we check that no *UIO* is assigned to a Process and an *Activity*. This check is necessary since the meta-model specifies those two associations as *XOR,* which is a constraint that cannot be directly mapped to an Ecore model. All other reference constraints that stem from the workflow meta-model are checked by the EMF validation framework itself, i.e. for example 1:1 relations.

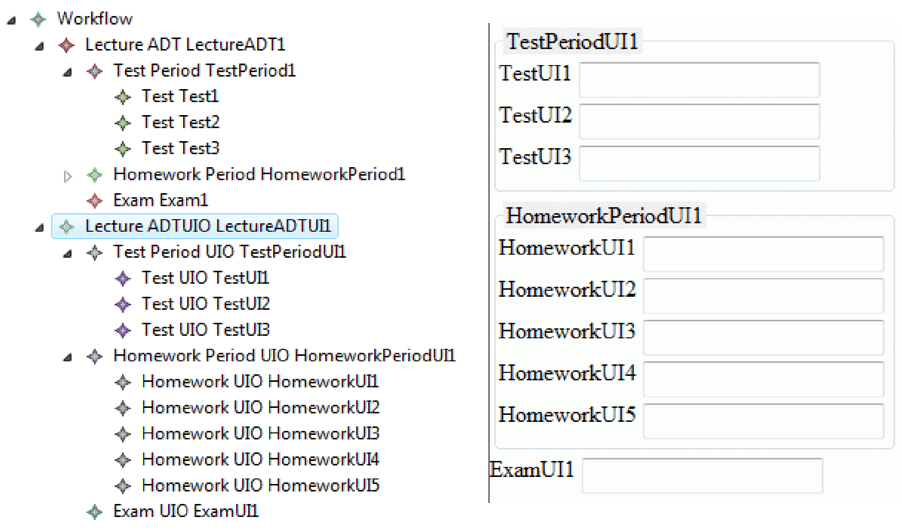
Besides those constraints there are rules which originate from the platform specific UI model and requirements towards the merging of business logic and user interface. For example there is a requirement that all *UIO* need to have an ID and another is that a groupbox has to have a caption defined.

After the model validation was successfully completed the source code generation is run. In our case Xpand's template engine is started with the root *Process* object and some appropriate templates [12] and functional extensions. Those extensions are specified in Xtend, another oAW language, and generate certain names and default XUL attributes for the source code.

In Figure 8 the resulting XUL UI is depicted on the right side. Left is the source Ecore model. As a slight deviation from the USE meta-model a root container type, named Workflow, was introduced within the EMF meta-model. The reason was purely technical. To reduce effort we always planned to use the generic EMF editor for our Ecore models. With that editor we needed a container object where to create the *UIO* instances. Instead of bending the workflow meta-model we decided to introduce a container class on top.

An import aspect to note about the user interface we created is that its layout is static in the first place. When the coupling to a runtime environment is eventually implemented we will be able to have the user interface become more dynamic.

In a first step we will enable or disable certain components depending of the state of their associated *Activity*. Also we might reveal certain *UIO*'s only if their activity was started. XUL in combination with a script language has the ability to this.



```
Workflow
  Lecture ADT LectureADT1
    Test Period TestPeriod1
      Test Test1
      Test Test2
      Test Test3
    Homework Period HomeworkPeriod1
    Exam Exam1
  Lecture ADTUIO LectureADTUI1
    Test Period UIO TestPeriodUI1
      Test UIO TestUI1
      Test UIO TestUI2
      Test UIO TestUI3
    Homework Period UIO HomeworkPeriodUI1
      Homework UIO HomeworkUI1
      Homework UIO HomeworkUI2
      Homework UIO HomeworkUI3
      Homework UIO HomeworkUI4
      Homework UIO HomeworkUI5
    Exam UIO ExamUI1
```

```
TestPeriodUI1
  TestUI1
  TestUI2
  TestUI3
HomeworkPeriodUI1
  HomeworkUI1
  HomeworkUI2
  HomeworkUI3
  HomeworkUI4
  HomeworkUI5
ExamUI1
```

Fig. 8. Ecore model and generate user interface

4 Conclusion

In this paper we have connected two fields that are relevant for the development of businesses process oriented software: Business process modeling and UI modeling and generation.

In the first part a declarative approach was presented to model flexible business process logic. The essential temporal relations of activities are expressed by that. A meta-model to support this approach was developed and presented. Using this meta-model a concrete process was specified, animated and visualized in the UML tool USE. Necessary constraints were presented and explained.

In the second part of the paper the UI aspect of workflows was covered. To develop a visible prototype of the UI it was useful to use existing technologies. The models of part 1 have been adapted to the EMF technology. Model to Model and Model to Text transformations have been used to derive a UI mockup from an instantiated workflow process. This user interface prototype was specified using XUL. Problems of connecting the USE implementation of declarative process models to UI modeling and generation have been discussed.

Both parts of the paper have been connected by using the same UML models and a throughout case study. Bidirectional dependencies of the business process and the user interface have been defined by OCL constraints.

References

1. OMG BPMN Specification 1.1: http://www.omg.org/docs/formal/08-01-17.pdf (visited July 19, 2009)
2. OMG UML Superstructure Specification v.2.1.2: http://www.omg.org/docs/formal/07-11-02.pdf, pp.295-418 (visited July 19, 2009)
3. Lecture of Abstract Data types, University of Rostock: http://wwwswt.informatik.uni-rostock.de/Lehre/Vorlesungen/VADT.html (visited May 9, 2009)
4. Brüning, J.: Declarative Workflow Modeling with UML Class Diagrams and OCL. In: BPSC2009, LNI-P 147, pp. 227-228, 2009.
5. A UML-based Specification Environment, University of Bremen: http://www.db.informatik.uni-bremen.de/projects/use/ (visited July 19, 2009)
6. van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow Patterns: Distributed and Parallel Databases, 14(3): 5-51, July 2003. http://www.workflowpatterns.com/documentation/documents/wfs-pat-2002.pdf (visited July 19, 2009)
7. Gogolla, M, Bohling, J., Richters, M.: Validating UML and OCL Models in USE by Automatic Snapshot Generation, *Journal on Software and System Modeling*, 4(4):386-398, 2005. http://www.db.informatik.uni-bremen.de/publications/Gogolla_2005_SOSYM.ps.gz (visited July 19, 2009)
8. OMG OCL Specification: http://www.omg.org/docs/ptc/03-10-14.pdf (visited May 9, 2009)
9. XUL, XML User Interface Language, http://developer.mozilla.org/en/XUL (visited July 19, 2009)
10. OpenArchitectureWare http://www.openarchitectureware.com/ (visited May 9, 2009)
11. Eclipse Modeling Framework Technology, http://www.eclipse.org/modeling/emft/ (visited July 19, 2009)
12. Wolff, A., Forbrig, P.: Deriving User Interfaces from Task Models. MDDAUI '09, Proc. of the IUI'09 Workshop on Model Driven Development of Advanced User Interfaces, CEUR Workshop Proc. 439. CEUR-WS.org, 2009.
13. Stahl, T., Völter, M.: Modellgetriebene Softwareentwicklung - Techniken, Engineering, Management, dpunkt.verlag, 2005
14. Pesic, M., Schonenberg, M.H., Sidorova, N., van der Aalst, W., et.al.: Constraint-Based Workflow Models: Change Made Easy. In: OTM 2007, LNCS 4103, pp. 77–94, Berlin, Springer, 2007.
15. Lonczewski, Schreiber; The FUSE System: an Integrated User Interface Design Environment; 1996
16. Szekely, P., Sukaviriya, P., Castells,P., Muthukumarasamy, J., Salcher, E.; Declarative interface models for user interface construction tools: the MASTERMIND approach; EHCI'95
17. Zhao, X., Zou, Y., Hawkins, J., Madapusi, B.; A Business-Process-Driven Approach for Generating E-Commerce User Interfaces; MoDELS 2007, Nashville

18 Pesic, M., van der Aalst, W.; DECLARE: Full Support for Loosely-Structured Processes, Proceedings of EDOC 2007, IEEE Computer Society, Annapolis, 2007

19 Schonenberg, M.H., Mans, R.S., Russell, N.C., et.al.; Towards a Taxonomy of Process Flexibility (Extended Version), BPM Center Report BPM-07-11, 2007 http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2007/BPM-07-11.pdf (visited July 19, 2009)