Electronic Communications of the EASST
Volume 36 (2010)

Proceedings of the Workshop on OCL and Textual Modelling (OCL 2010)

Extending ASSL: Making UML Metamodel-based Workflows executable

Jens Brüning, Lars Hamann, Andreas Wolff

12 Pages

Guest Editors: Jordi Cabot, Tony Clark, Manuel Clavel, Martin Gogolla
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Extending ASSL: Making UML Metamodel-based Workflows executable

Jens Brüning[1], Lars Hamann[2], Andreas Wolff[1]

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

Abstract. ASSL is a language that enables UML developers to test and certify UML and OCL models [5]. Snapshots of system states are semi-automatically created and main parts of the UML action semantics is implemented by the language. Its interpreter is the well-known UML modeling tool USE. The article proposes a number of language extensions to ASSL. These include (sub) procedure calls and pre- and postcondition checks on entering and exiting of operations using OCL. The paper motivates the need for these extensions as well as their usage and development along the problem of metamodel-based execution of workflow models. Executable workflow models, driven by ASSL procedures, are introduced in detail to present the usage of ASSL and our extensions.

Keywords: Model validation, Model execution, A Snapshot Sequence Language, Workflow Metamodels

1 Introduction

The *UML-based Specification Environment* (USE) [6] is a tool that can generate UML object diagrams from class diagrams manually or semi-automatically. These derived object diagrams can be seen as snapshots of a running system. USE enables a developer to specify declarative OCL constraints in class diagrams. During runtime, these constraints, like e.g. invariants for system states or pre- and postconditions for UML operations, are permanently checked against the current snapshot.

USE provides a language called *A Snapshot Sequence Language* (ASSL) [6]. ASSL has the ability to semi-automatically generate object diagrams. In this process all possible assignment combinations of objects and variables are attempted to find a stable state which satisfies every defined constraint [5]. If no assignment meets all OCL invariants the ASSL generation procedure finishes without results. For finding valid snapshots the special command *Try* provides the possibility to assign values to ASSL variables that are further used for generating valid snapshots. The special command *Any* assigns any value of a set to a variable. To generate

object diagrams ASSL procedures must represent imperative specifications. It implements a large part of the UML action semantics including the creation or deletion of objects and links and the setting of attribute values in UML object diagrams. This is crucial for testing as well as executing UML models. The approach of this paper relies on those operations as basis for executing workflow models.

ASSL has been implemented in combination with the parser generator ANTLR [11]. However, this article does not focus on implementation details. It rather explains how we use the extensions in the context of UML metamodel-based workflow execution. We are confident that there is a number of further promising applications of the proposed ASSL extensions. Especially the area of model testing and certification in connection with the unique commands *Try* and *Any* for semi-automatic snapshot generation seems to bear good prospects for use.
The workflow modeling and execution approach is a new application for the USE tool and ASSL. The presented approach comprises of a declarative and an imperative part, while the focus of this article is on the imperative part. Our approach enables us to express the workflow patterns presented in [12]. In contrast to established workflow languages like EPCs, UML activity diagrams or BPMN the modeling approach has a flexible background driven by design principles. All execution sequences of the process model are allowed if they are not forbidden by OCL constraints. In contrast, the established languages uses a more Petri net-like modeling approach in which only the allowed execution flows are determined. The developer defines action sequences that may restrict the user too much while executing the workflow [13]. In our view, the work presented in this paper is a new direction in the context of workflow languages with a declarative metamodel-based approach.

The rest of the article is structured as follows. Section 2 introduces our metamodel for workflows. We model an example workflow on basis of that. Also we present a design time plugin to USE that captures workflow models as ASSL instantiation procedures. This way we can reuse these models at runtime. Section 3 introduces the workflow plugin that presents the workflow to the developer for interaction. We also go into details about ASSL and our ASSL extensions as they are the basis for workflow executions. A UML sequence diagram shows the relevant ASSL procedure calls. Section 4 discusses related work and Section 5 concludes the work.

2 Workflow modeling with UML metamodel

In this section we introduce the metamodel for workflows and demonstrate how workflows are modeled by means of the USE tool. We introduce a design time plugin of USE that persistently stores the workflow models for later reuse by generating ASSL procedures.

2.1 UML metamodel for workflows

An earlier version of our metamodel for workflows was introduced in [2]. Figure 1 shows an extended version that now supports all original 20 workflow patterns [12]. Besides the class model, the metamodel contains of OCL invariants and pre- and postconditions to express the semantics of most metamodel elements declaratively. Behaviors of the temporal or causal relations are also expressed imperatively. This particular part of the metamodel is implemented

as ASSL code. It will be explained in section 3 and is the main contribution of this paper. The following is intended as an overview to roughly explain the metamodel, as this is the key to understanding the semantics implemented in ASSL.

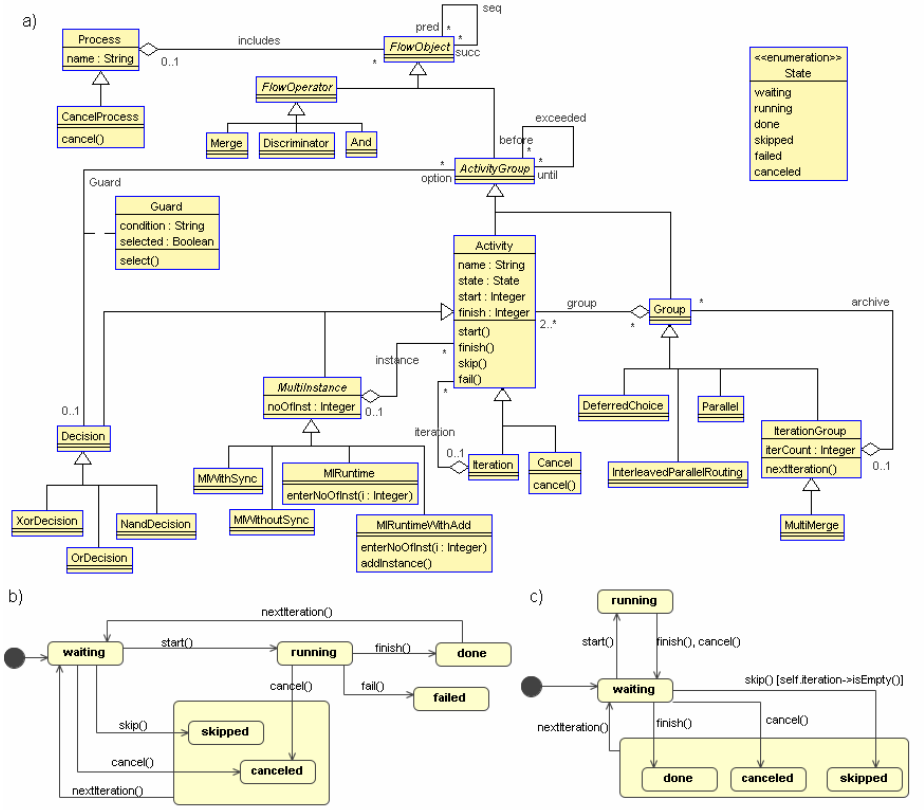

Fig. 1. a) The metamodel shown as UML class diagram b) UML state diagram showing life cycle of objects of the class *Activity* c) object life cycle of the objects of the class *Iteration*

An analysis of how far our metamodel supports the workflow patterns is beyond the scope of this paper. However, a first discussion of this matter can be found in [2].

The pivotal class of the metamodel is *Activity*, shown in the center of Figure 1a). Enumeration *State* lists the possible execution states of an activity. Figure 1b) shows a life cycle of an activity as UML state diagram. In our work state transitions are expressed by OCL pre- and postconditions. For instance the precondition of the *start()* operation requires the object to be in the state *waiting*. Its postcondition consequently assures that the state has changed to *running*. States of activities can be changed by calling operations of the classes *Activity*, *IterationGroup*, *Cancel* and *CancelProcess*. They are implemented with ASSL.

Note that not all operations changing an *Activity's* state are declared in that class. For example, an object of *IterationGroup* can initiate another iteration through the operation *nextIteration()*. This would store all execution data of the current iteration to the *archive* and reset all included activities to *waiting*. Class *Activity* itself does not directly provide an operation for resetting its instances' state.

The state diagram of the class *Iteration* is shown in Figure 1c). It differs from *Activity's* (see Figure 1b) in that new iterations can be started after one is finished without resetting the

activity. If an *Iteration* object is in the state *running* and the operation *finish* is called a new iteration can be started directly by calling *start* again. The behavior of *Iteration* is described more deeply in [7].

Operation execution can have side effects on other activities, depending on causal or temporal relation between them. ASSL procedures implement those. If for example an activity starts and this activity is member of a *DeferredChoice* group all other activities of that group are skipped. Thus, the other activities cannot be started anymore and the choice was done implicitly. Explicit decisions are expressed through the class *Decision* and its subclasses. The criteria to select follow-up activities here are declared in the association class *Guard*. The selection is user-driven and executed at runtime. This process will be discussed in subsection 3.1.

2.2 Workflow model shown as UML object diagram

Figure 2 exemplifies the use of the workflow metamodel for the case of a medical emergency process. It essentially shows a screenshot of the USE tool, which provides the modeling environment and thereby an abstract syntax for workflow models.
The main *Process* object is arranged topmost left in Figure 2. It serves as root object to which all other model objects are connected; either direct or indirect through transitive associations. There is an OCL operation to collect all these elements through calculating the transitive closure. The operation also is part of the metamodel but not explicitly listed in Figure 1a).

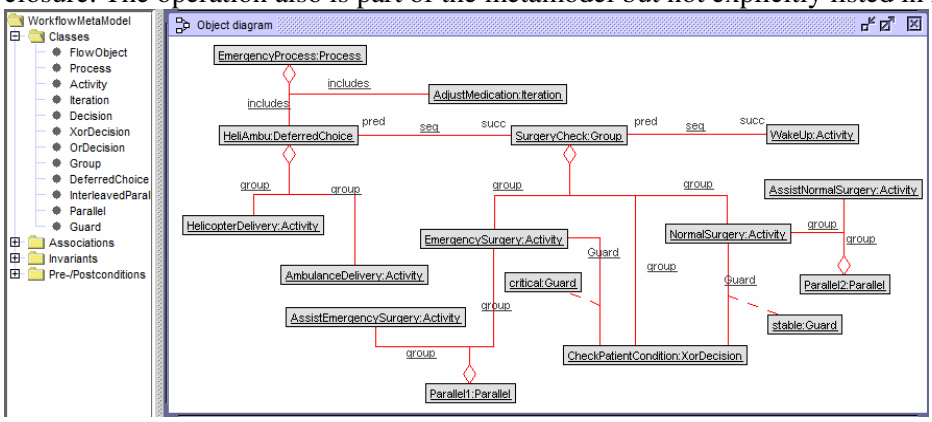

Fig. 2. Example process model with the abstract syntax provided by USE

The emergency process begins with the delivery of the patient. She can either be transported by helicopter or ambulance. For this initial part of the workflow the hospital staff is not responsible to decide what transport type should be taken. Therefore both available transportation activities are modeled in a *DeferredChoice* relationship [2]. After the patient has arrived at the hospital, she has to be checked whether she has to be operated immediately or if there is time to prepare a normal surgery. This check is done by a doctor at the hospital. Depending on its decision, an immediate or a normal surgery takes place. The *NormalSurgery* as well as the *EmergencySurgery* is assisted by nurses and an anesthetist. This fact is modeled by *Assist* activities that are related together with the respective *Surgery* activities in *Parallel* relationships. Afterwards, the patient wakes up which has to be observed by the hospital staff and is represented as an activity in the workflow model. During the whole process the

medication of the patient proceeds and has to be continuously documented. This fact is modeled by *AdjustMedication* that activity is an *Iteration* and thus can be executed several times during process execution. No further temporal constraints to other process fragments are to be observed here.

2.3 USE design time plugin

USE is capable of storing the current snapshot of models. But USE is not able to duplicate a snapshot in the object diagram. In the following a process and a developed plugin for USE is proposed to enable the user to instantiate a workflow model several times. Thus, instances of a process model can run in parallel after they have been instantiated. For this purpose a specialized plugin to USE had to be developed. We call it "design time plugin" as this describes the time when it is applied in contrast to the "runtime plugin" that we introduce in subsection 3.1. It persistently stores the workflow model as ASSL instantiation procedures.

A process developer will invoke the plugin after she completed modeling the workflow. The plugin provides a dialog to choose an ASSL file into which the ASSL instantiation procedure is generated. Listing 1 shows parts of an ASSL instantiation procedure that was generated from our sample workflow model. When executed, the procedure recreates the objects and associations of the model shown in Figure 2. Furthermore, the states of the *Activity* objects are set to the initial state *waiting* according to the state diagrams of the metamodel of Figure 1.

Listing 1. Excerpt of an ASSL workflow instantiation procedure

```
procedure instantiateEmergencyProcess()
var a1:Activity, a2:Activity, d1:DeferredChoice ...;
begin
  a1 := Create(Activity);
  [a1].name := ['HelicopterDelivery'];
  [a1].state := [#waiting];
  a2 := Create(Activity);
  [a2].name := ['AmbulanceDelivery'];
  [a2].state := [#waiting];
  d1 := Create(DeferredChoice);
  Insert(group, [d1], [a1]);
  Insert(group, [d1], [a2]);
  ...
end;
```

To use these instantiation procedures for executable workflows, another USE plugin was developed, the "workflow runtime plugin". Among other things, in this plugin a user can select the desired ASSL file and the included workflow instantiation procedure to invoke it and consequently instantiate the workflow.

Original ASSL commands, as presented in [5], are sufficient for this purpose. Only some procedures of the workflow execution require ASSL language extensions, which will be discussed in section 3. One characteristic of ASSL is the use of square brackets to enclose OCL expressions. They may contain and use ASSL variables declared and initialized earlier in that ASSL procedure. OCL expressions may become quite complex as, e.g., shown in Listing 3.

3 Workflow model execution

This section introduces the execution of workflow models using the workflow runtime plugin. This plugin presents a workflow instance to its user in an appropriate way and provides a GUI to invoke the ASSL procedures. Subsection 3.2 introduces the ASSL extensions that provide the basis to implement the execution semantics of the workflow models. Subsection 3.3 discusses the ASSL implementations for model execution. A non-plugin feature, but nevertheless very handy is USE's ability to log the ASSL procedure executions and present them as a sequence diagram. This is demonstrated in subsection 3.4.

3.1 Workflow runtime plugin

Figure 3 is a screenshot of the workflow runtime plugin presenting an instance of the example workflow of Figure 2. The activity list uses colors to indicate the state of each activity. The workflow plugin distinguishes between *waiting* and *enabled* activities. The state *enabled* is not provided in the state diagrams of Figure 1 to keep the approach quite simple and manageable. *Enabled* activities appear in a light green color. *Waiting* activities that are forbidden to be executed by OCL constraints are colored in a darker green color. The workflow plugin checks the *enabled* property of activities in a preprocessing step.

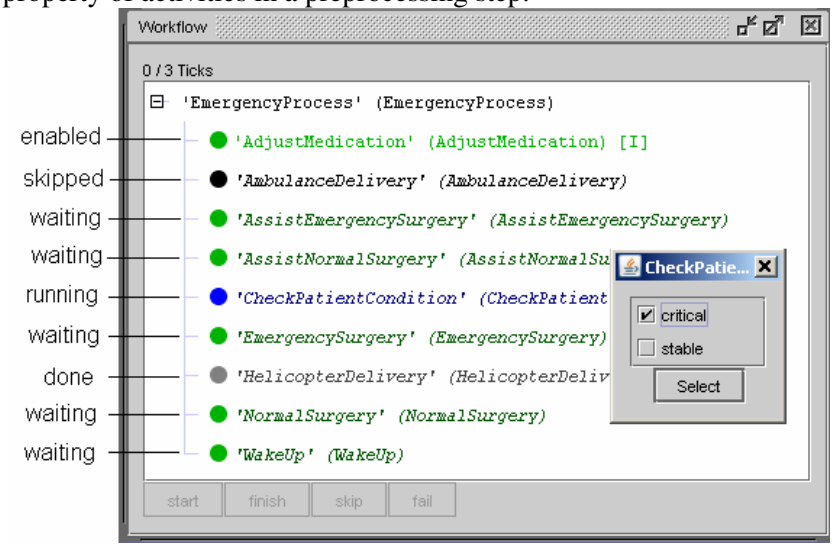

Fig. 3. Workflow runtime plugin showing a workflow instance

Currently, in the scenario of Figure 3 the activity *CheckPatientCondition* is running which is expressed by the blue color. This activity is a *Decision*. This decision is to be made by the user, thus, a further interactive window is generated by the workflow plugin to request the user's decision interactively. The available options or alternatives and its selection criteria are declared in the workflow model. Having selected the appropriate criterion, the *Decision* activity ought to be finished by clicking on the corresponding button at the bottom of Figure 3. Consequently the plugin invokes the ASSL *finish* procedure on the selected activity. Those

buttons represent the *Activity* operations as shown in the metamodel of Figure 1. ASSL is used to implement those operations. Details on this matter are in subsection 3.3.

3.2 ASSL language extensions

Table 1 lists our ASSL extensions, primarily new commands. *ASSLCall* provides a command to invoke procedures. This provides the ability for recursive procedure calls.

Table 1. New ASSL commands

New ASSL commands	Explanations
ASSLCall <proc-name> (<*arguments*>);	Calling another ASSL procedure (in the same ASSL file). The arguments are separated by comma.
OpEnter <OID> <op-name> (<arguments>);	Enters an operation with the *op-name* in the context of the object identified by its *OID*. Arguments are separated by comma. USE checks the OCL preconditions.
OpExit;	Exits the running operation that lies on top of the (operation) call stack and USE checks the OCL postconditions.

OpEnter steps into the given operation of a certain specified object. *OpEnter* only checks the OCL preconditions of the declared operation and object, but is not executing the operation. Instead it pushes the operation on top of the general (operation) call stack which is administered by USE.

Command *OpExit* specifies that the given operation is finished and the OCL postconditions ought to be checked. The developer can neither declare an object nor an operation to exit. The USE environment checks the postconditions of the operation lying on top of the call stack. This is the last operation that was started with an *OpEnter* command before.

3.3 ASSL procedures for the workflow model execution

Several ASSL procedures implement the base operations of the workflow metamodel classes. Some operations get overridden by specified implementations in subclasses. For example the *start()* operation of *Activity* behaves differently than the *start()* operation of *Iteration*.
Overriding operations is achieved by ordering the procedures in the ASSL file in a certain way. Procedures with more specialized types as arguments are declared before the ones with more general types. The semantics of finding a procedure with a fitting signature is as follows. By invoking an ASSL procedure USE parses the ASSL file top-down. The first procedure with a signature fitting to the called procedure name and passed on arguments, is selected for execution. Thus, we would order a procedure *start(i:Iteration)* before *start(a:Activity)*. Then, if *start()* is invoked with an *Activity* object the first signature would not fit but the second one does, so consequently *start(a:activity)* is used.

Ordering the procedures the other way round implies that *start(a:Activity)* also fits with *Iteration* objects because of the substitution principle [8]. Consequently, USE would never execute *start(i:Iteration)* with *Iteration* objects.

Listing 2 shows an excerpt of the ASSL *start(a:Activity)* procedure and demonstrates the use of the ASSL language extensions of Table 1. At the beginning of this procedure the *OpEnter* command causes the preconditions to be checked. Then a change of the activity's state is specified, from *waiting* to *running*. Following up, side effects on other activities are implemented. All activities related within the same *DeferredChoice* group are skipped and all *Parallel* activities are started. Finally, the *OpExit* command initiates the postconditions checks.

Listing 2. Excerpt of the ASSL start procedure for class Activity

```
procedure start(a:Activity)
var setA:Set(Activity);
begin
  -- checking precondition of operation
  OpEnter [a] start();
  -- changing state to running
  [a].state:=[#running];
  for gr:Group in [a.group->asSequence] begin
    -- skipping all deferred choice activities
    if  [gr.oclIsTypeOf(DeferredChoice)] then begin
      setA := [gr.activity
                    ->select(a2|a2.state=#waiting)];
      for a2:Activity in [setA->asSequence] begin
        ASSLCall skip([a2]);
      end;
    end;
    -- starting all parallel activities
    if [gr.oclIsTypeOf(Parallel)] then begin
      setA := [gr.activity->select(a2|
                 a2<>a and a2.state<>#running)];
      for a2:Activity in [setA->asSequence] begin
        ASSLCall start([a2]);
      end;
    end;
  end;
  ...
  OpExit;
end
```

As discussed earlier the ordering of ASSL-procedures in a command file is of importance. Consequently, procedure *finish(d:Decision)* precedes *finish(a:Activity)* in the ASSL file. A call *finish(CheckPatientCondition)* (see workflow model of Figure 2) matches the ASSL procedure for *Decisions* and USE would select that implementation for execution. Listing 3 declares the behaviour of it. A special characteristic of that procedure is that it causes side effects on subsequent activities. Non-selected activities and groups of activities are skipped because they must not be executed afterwards. In contrast, selected activities are enabled for execution.

Listing 3. Excerpt of the ASSL finish procedure for class Decision

```
procedure finish(d:Decision)
var setAG:Set(ActivityGroup), setA:Set(Activity);
begin
  OpEnter [d] finish();
```

```
    [d].state := [#done];
    -- get all non-selected activities and groups
    setAG:=[d.option->select(a|
            a[option].guard.selected <> true)];
    -- collect all non-selected activities
    setA:=[setAG.oclAsType(Activity)
           ->select(isDefined())
           ->union(setAG.oclAsType(Group)
           ->select(isDefined()).activity)->asSet()];
    -- skip all non-selected activities
    for a:Activity in [setA->asSequence] begin
      ASSLCall skip([a]);
    end;
    ...
    OpExit;
  end;
```

3.4 UML sequence diagram showing the ASSL procedure calls

Figure 4 shows a scenario of a workflow execution. USE has logged the ASSL commands *OPEnter* and *OPExit* as they occurred and presents the chronology of executed calls as a sequence diagram.

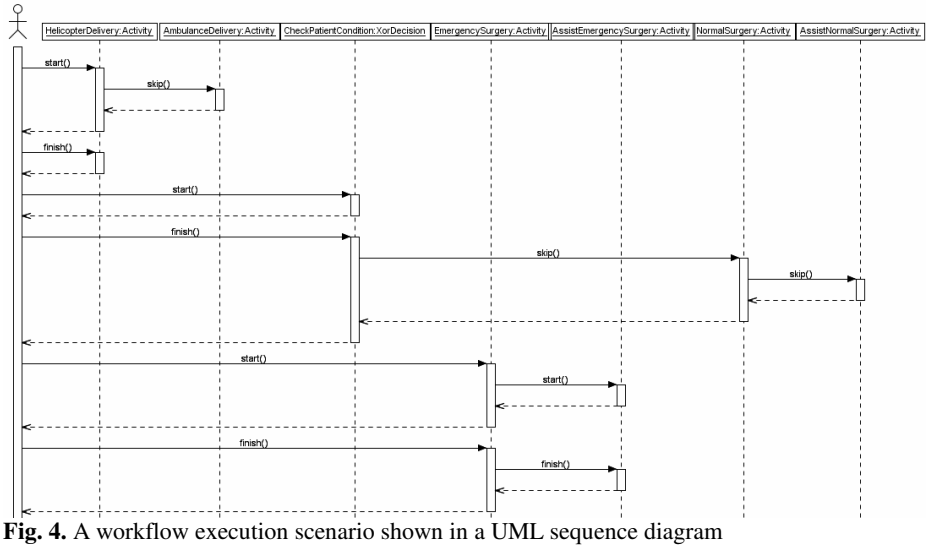

Fig. 4. A workflow execution scenario shown in a UML sequence diagram

This scenario, started with a *HelicopterDelivery*. As shown in the *start()* procedure's implementation all activities that are related in a *DeferredChoice* were skipped implicitly. According to the ASSL implementation USE skipped *AmbulanceDelivery* for this case. After arriving at the hospital, a doctor has checked the patient. Finishing that decision activity caused any non-selected activity to be skipped. This semantic is implemented in the ASSL *finish()* procedure shown in Listing 3. Here, *NormalSurgery* was skipped. Calling the ASSL *skip()* procedure has the consequence that all parallel activities are skipped, too. Thus, *AssistNormalSurgery* is also skipped. The same applies for the *start()* and *finish()* operation of activity *EmergencySurgery* and *AssistEmergencySurgery*.

4 Related work

There exist several other languages that implement the UML action semantics, a well-known example is QVT [9]. Kermeta [1] is an open source metamodelling environment that has been designed as an extension to the metadata language EMOF [9] with an action language for specifying semantics and behavior of metamodels. Parallel to this work of extending ASSL, the OCL-based imperative programming language SOIL (Simple OCL-based Imperative Language) has been developed [3] that can also be interpreted by the USE tool. As mentioned in the introduction, ASSL can be used for semi-automatically generate snapshots of object diagrams in contrast to the languages listed above.

For workflow modeling some metamodel-based approaches exist like for example the EMF metamodel-based Bflow [7] tool in which Event-driven Process Chains (EPC) are used as workflow language. Bflow checks static properties of the workflow models but lacks execution semantics. Execution semantics used with a metamodel approach for UML activity diagrams is presented in [4]. Following the UML specification [10 (section 12)], this approach uses a Petri net-like token flow semantics. In contrast, the approach presented in this paper is, to our knowledge, the only one that uses a pragmatic UML metamodel-based declarative approach to express the workflow patterns and execute the workflow models on basis of imperative ASSL code.

5 Conclusion

This article presented extensions of the ASSL language: (Sub-) procedure calls as well as precondition checks on entering operations and postcondition checks on exiting are now possible with ASSL. The ASSL language extensions were introduced in the context of the metamodel-based workflow modeling and execution.

The workflow approach comprises a declarative part with OCL invariants, pre- and postconditions and an imperative part with ASSL procedures for the model execution. USE provides a modeling and a runtime environment for workflows. A newly developed workflow plugin to USE presents the workflow instance to the developer in an appropriate way. By clicking on buttons that represent operations of the metamodel, the user invokes ASSL procedures implementing the selected activity. Thus, the developer can execute scenarios and test dynamic control flow properties of its workflow models. USE logs the scenarios as a sequence diagram to visualize the workflow executions for further analysis.

References

1. Baudry, B., Nebut, C., Le Traon, Y.: Model-driven Engineering for Requirements Analysis, 11th Enterprise Distributed Object Computing Conference (EDOC 2007), IEEE International (2007)
2. Brüning, J., Gogolla, M., Forbrig, P.: Formally Checking Workflow Properties Using UML and OCL. 9th International Conference on Perspectives in Business Informatics Research (BIR 2010), Springer, LNBIP vol. 64 (2010)

3. Büttner, F., Gogolla, M.: Reusing OCL in the Definition of Imperative Languages. http://www.db.informatik.uni-bremen.de/publications/intern/fb_mg_soil_2010.pdf (accessed: 04/01/2011)
4. Engels, G., Soltenborn, C., Wehrheim, H.: Analysis of UML Activities Using Dynamic Meta Modeling. Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS 2006), Springer, LNCS vol. 4468 (2007)
5. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL models in USE by automatic snapshot generation. Software and System Modeling, 4(4):386–398 (2005)
6. 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)
7. Kühne, S., Kern, H., Gruhn, V., Laue, R.: Business process modeling with continuous validation, Journal of Software Maintenance and Evolution: Research and Practice, Volume 22, Issue 6-7, pages 547–566 DOI: 10.1002/smr.517 (2010)
8. Liskov, B., Wing, W.: A Behavioral Notion of Subtyping. ACM Transactions on Programming Languages and Systems, 16:1811-1841 (1994)
9. Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. OMG document formal/08-04-03, (2008) http://www.omg.org/spec/MOF/2.0/PDF (visited: 04/01/11)
10. Object Management Group. Unified Modeling Language (UML) version 2.3. OMG document formal/2010-05-05 (2010) , http://www.omg.org/spec/UML/2.3/Superstructure/PDF (accessed: 04/01/11)
11. Parr, T.: The Definitive ANTLR Reference Guide: Building Domain-specific Languages, Pragmatic Programmers (2007)
12. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow Patterns. Distributed and Parallel Databases, 14(3):5-51 (2003)
13. 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.