Modeling and Executing ConcurTaskTrees using a UML and SOIL-based Metamodel

Jens Brüning
University of Rostock
Albert-Einstein-Str. 22
D-18051 Rostock
Jens.Bruening@uni-rostock.de

Martin Kunert
University of Rostock
Albert-Einstein-Str. 22
D-18051 Rostock
Martin.Kunert@uni-rostock.de

Birger Lantow
University of Rostock
Albert-Einstein-Str. 22
D-18051 Rostock
Birger.Lantow@uni-rostock.de

ABSTRACT
In this paper, we present a formal metamodel-based approach for modeling and executing ConcurTaskTrees (CTT). CTT is a modeling language for tasks that represent hierarchical tree-like, structured workflow models. In the metamodel soundness properties as well as operational semantics of the workflow language are captured. The models are created with an abstract syntax in the *UML-based Specification Environment* (USE). Thereafter, they can be executed in the same tool by calling operations from the metamodel that are implemented in the *Simple OCL-based Imperative Language* (SOIL). A plugin for the USE tool has been developed to give an appropriate interface to the user who can test dynamic control flow properties of the models with it.

Keywords
Metamodeling, Taskmodeling, UML, OCL, SOIL, Model execution.

1. INTRODUCTION
Business process modeling gets more and more important with the increasing complexity and automation of business processes in companies and organizations. They are used to document, restructure and optimize the processes. Furthermore, requirements for software and computer services that support the business processes are captured in these models.

Nowadays, workflows are frequently modeled in a non-hierarchical, flat way in EPCs, UML activity diagrams or BPMN. Although subprocesses can be defined, the hierarchical modeling is not an integral part of these languages and can only be integrated in an unnatural, difficult to handle way. Having several hierarchies would mean managing several models and inserting a new hierarchy would mean creating new models. Whereas hierarchy is an integral part in function trees that are used in the ARIS method and toolkit [21] for business process modeling. Hierarchical supply and value chains are a common feature in the domain of business modeling. Task models are used in the HCI domain for capturing the hierarchical character of tasks and get expressed in a tree-like notation.

ConcurTaskTrees are widely used in the community of model-based user interface development. They represent hierarchy-oriented workflow specifications. By task decomposition the inner nodes of the task trees represent user-centered, easy-to-use goal specifications [10]. The leaf tasks are executable activities. Within the hierarchical structure the control flow is specified by unary and binary temporal operators from the process algebra language LOTOS [4].

The benefits of trees to be used with workflow models are widely analyzed and accepted [13,14]. The tree model is used to express structuredness within the workflow models rather than goal modeling. But both properties can be connected with tree models.

The approach presented in this paper uses a UML metamodel along with the tool USE [1]. USE checks static properties of the workflow models during the modeling process by observing OCL invariants. The modeler gets quick feedback of identified problems and the involved modeling elements are immediately presented to her.

There are even more benefits to UML workflow metamodels with respect to dynamic properties. They provide means to define execution semantics. OCL invariants are used for system states and pre- and postconditions for operations. They describe the causal or temporal relationships between the modeling elements in a platform independent way. During execution of the workflow model, the operational semantics is interpreted and disallowed flows of a process are forbidden. Furthermore, enabled activities can be identified and they are indicated in the GUI to guide the user through the workflow model execution.

The main goal of the paper is to provide a formal semantics for CTTs that is executable and allows a model analysis using OCL at design time. Having the operational semantics in the metamodel we show that the models can be executed in a prototypical implementation in connection with the USE tool.

The remainder of the paper is structured as follows. The temporal operators and CTT-language itself is more deeply introduced in section 2. We present our metamodel for CTT in section 3. Also, some structural soundness properties and the operational semantics that is expressed in the metamodel are presented there. The workflow modeling part and the execution in the USE tool is presented in section 4. We present related work in section 5 and conclude the paper in section 6.

2. Temporal Operators and Task Decompostition in CTT Models
Firstly, we introduce the temporal CTT-operators in subsection 2.1. All the siblings in the task tree have to be connected with binary temporal operators in a chain. Every task can be notated with a unary one as well. We present the task decomposition with an example in subsection 2.2. It is used as ongoing example in this paper to present the metamodel-based approach as well.

2.1 Temporal Operators
First of all, for creating the task model, the tree structure has to be established by task decomposition. The overall goal is denoted in the root node and it is recursively decomposed into subgoals until

the action level is reached in the leaf tasks (see section 2.2). All the siblings in the task tree have to be connected with binary temporal operators. They are listed with the CTT syntax and their informal operational semantics in Table I from number 1 to 10.

As stated in Table I the *Concurrency* and the *Enabling* operators have each another operator to specify that information is exchanged between the two connected tasks. But the kind of data cannot be declared in detail.

TABLE I. TEMPORAL OPERATORS IN CTT

Operator Description	***CTT-syntax***	***Operational Semantics***
1. Choice	A [] B	Either *A* or *B* has to be executed.
2. Concurrency	A ||| B	*A* and *B* has to be executed concurrently.
3. Concurrency with information exchange	A|[]|B	*A* and *B* are executed concurrently and information is exchanged between them
4. Disabling	A [> B	*A* is disabled by *B*. *A* has to be in the state enabled, running or done to be disabled by *B*.
5. SuspendResume	A |> B	*B* suspends the execution of *A*. This can happen several times during the execution of *A*.
6. OrderIndependence	A |=| B	Either *A* or *B* including their subtasks have to be fully executed before the other task can be started.
7. Enabling	A >> B	*A* has to be finished before *B* can be started.
8. Enabling with information passing	A[]>>B	Additionally, *A* provides information for the execution of *B*.
9. Iteration	A*	*A* can be executed 0-n times in iteration
10. Option	[A]	*A* can be executed optional

The binary operators have specific binding strength to specify the binding semantics that are needed if more than three sibling tasks exist in one hierarchy in the tree model. The binding strength is indicated downwards in Table I with the highest priority connected to the choice operator. The least priority is consequently connected to the enabling operator. To show the binding priority we take the following example process term: *A >> B [] C |> D*. It is interpreted as *A >> ((B [] C) |> D)* after the priority specifications of Table I.

Because brackets do not fit into the graphical tree-like syntax of CTT, the binding strength of the operators are needed. If the prescripted priorities of the operators do not fit to the flow logic you want to model, another artificial tree node has to be inserted (see [4]). In the model of Figure 1 this is done with the *TransSurg* task, which will be deeper introduced in subsection 2.2.

Apart from binary operators, every task can be additionally denoted with an unary operator (see numer 9 and 10 in Table I). Following the semantics of these operators, the task is either iterative or optional.

2.2 Task Decomposition and CTT Example Model

In this subsection we introduce an emergency process example taken from the hospital domain. The CTT model of Figure 1 represents the emergency process that is so called in the root node of the task tree.

The root task has two subtasks. *TransSurg* represents the compound task for the transportation and surgery of the patient. Its subtasks execution can be suspended anytime by the task *AdjustMedication*. This is a leaf node and thus can be executed. During its running period, the medication takes place and should be documented.

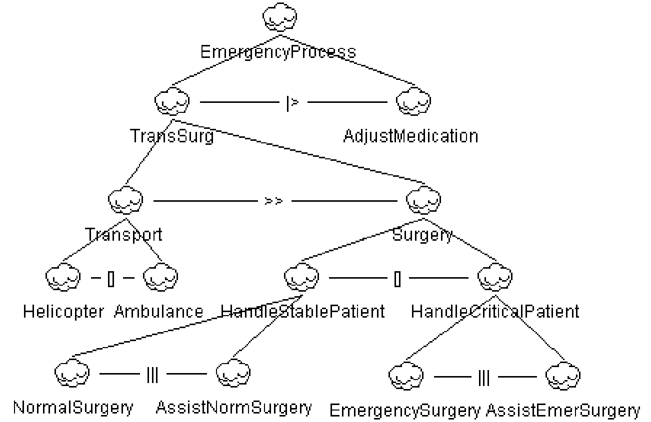

Figure 1: An Emergency Process modeled in a CTT model

As mentioned before, we had to insert the *TransSurg* as additional task. Thus, the range of the *SuspendResume* operator expands over the *Transport* and *Surgery* tasks. If we had specified these three tasks as siblings in a chain as follows: *Transport* >> *Surgery* |> *AdjustMedication* the control flow would have been interpreted as *Transport* >> (*Surgery* |> *AdjustMedication*)

The *Transport* task is decomposed into *Helicopter* and *Ambulance* that represents the way of transportation and arrival of the patient at the hospital. These two tasks are related in a choice dependency to each other. Thus only one task of them can be executed and the other is skipped consequently.

After the arrival at the hospital the process continues with the surgery. If the patient is in a critical condition that is represented by the task *HandleCriticalPatient*, an emergency surgery has to be done. In this case there is no time to prepare a surgery in a sterile environment.

The alternative is specified by the task *HandleStablePatient* in which the situation is not critical and a *NormalSurgery* can be prepared in the operation room. These two tasks are related in a choice relationship. Both *Surgery* tasks are specified together with *Assist* activities that are related in a *Concurrency* relationship to the surgery activities. They take place in parallel and represents the task the nurses or anesthetist are responsible for.

The emergency process is entirely done when the *Surgery* and *Assist* activities are executed. Following the operational semantics of the *SuspendResume* operator, the suspending task *AdjustMedication* is not further enabled for execution if the previous task is already done.

3. ConcurTaskTree Metamodel

The metamodel of CTT comprises the class diagram that is introduced in subsection 3.1. OCL invariants that are presented in subsection 3.2 express soundness properties. The operational semantics is based on state charts for task life cycles that are introduced in subsection 3.3. In subsection 3.4 the operational

semantics are implemented with SOIL [2]. The *start()* operation is introduced by example to show the use of SOIL in the metamodel.

3.1 Class Diagram

In Figure 2 the UML class diagram is pictured. We see the abstract class *Task* that has a reflexive associationclass *TempOp*. The multiplicity 0..1 is used for this and specifies the binary temporal relationships between the sibling tasks. *TempOp* is kept abstract itself. Thus, its concrete subclasses represent the binary temporal relationships between the sibling tasks in CTT.

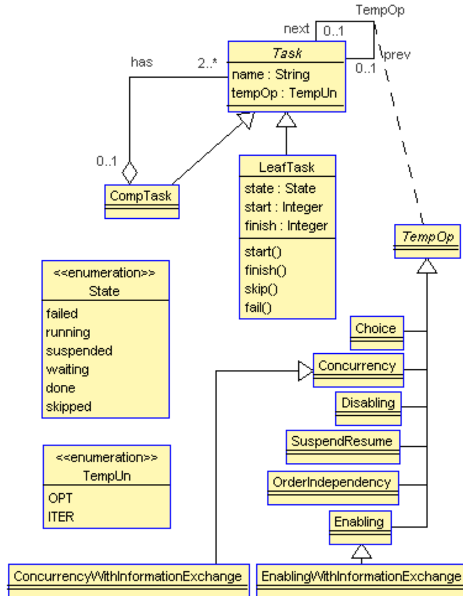

Figure 2: The class diagram for the CTT metamodel

Within the concrete associationclasses from Choice to Enabling the operational semantics can be specified with OCL invariants in a platform independent way. With these invariants the correctness of the SOIL implementation (see subsection 3.4) can be ensured similar to the Design by contract principle [10]. The *Enabling* class for example has an invariant that states if the *next* task is *running* the previous one has to be *done*, *skipped* or *failed*.

The unary operators *Iteration* and *Option* of CTT (ref. Table I) are expressed with the enumeration *TempUn*. There is an attribute *tempOp* referring to that type in the class *Task*. Thus, if that attribute is set the appropriate task has the corresponding property. Otherwise the attribute is *Undefined* which means that the task is normal and has the normal task life cycle (see subsection 3.3).

The task decomposition for creating the tree structure is enabled by the association *has* that connects a complex *CompTask* with its at least two subtasks. Through this association both composed and leaf tasks can be connected as subtasks.

The class *LeafTask* represents the executable leaf tasks in the task model. It has operations that represent the interface of the task objects or *work items* to the user. She can invoke *start*, *finish*, *skip* and *fail* on these objects during runtime. The attribute *state* stores the execution state of the task that is used during runtime.

3.2 CTT Structural Soundness Properties expressed with OCL Invariants

The structure for CTT task models or structural soundness properties can be expressed by OCL invariants. For example the tree structure produced by the association *has* can be expressed by invariants. This is done implicitly by the ones given in Listing 1.

Invariant *taskStructure* belongs to the associationclass *TempOp* and ensures that the by temporal relations (*TempOp*) connected tasks must have the same parent's task. The next invariant *startAndEndTasksExist* expresses that both a start and an end task have to exist in one subtree hierarchy. The start task has no predecessor and the end task has no successor that is connected by the association *has* in that subtree hierarchy. Thus, only chains and no cycles of temporal operators are allowed to relate tasks to each other.

As already stated in subsection 3.1, also the operational semantics can be expressed by invariants but this matter we omit here and only present the SOIL implementation in subsection 3.4.

Listing 1. OCL-invariants for the structural soundness properties

```
context TempOp inv taskStructure:
  prev.compTask = next.compTask

context CompTask inv startAndEndTasksExist:
task->select(prev.isUndefined())
  ->size()=1 and
task->select(next.isUndefined())
  ->size()=1
```

3.3 State Charts

We use state chart diagrams as the basis to express the operational semantics of the task models. The states of the state diagrams of Figure 3 are introduced in the enumeration *State* of the class diagram in Figure 2. In contrast, the original CTT-language uses the event-based process algebra LOTOS [4].

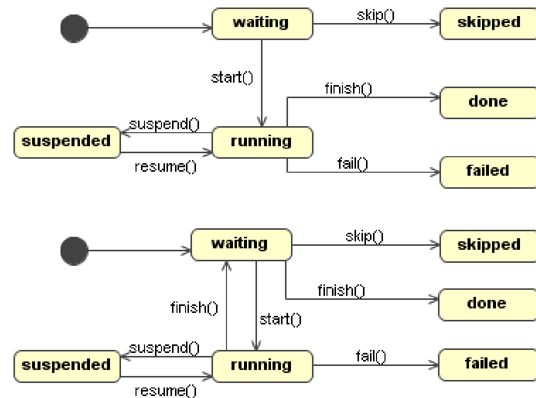

Figure 3: Normal and iterative task life cycles

The task life cycles start in the state *waiting*. After calling the *start()* operation on a leaf task, it changes to the state *running*. The state chart diagrams are emulated by pre- and postconditions connected to the operations listed in the class *LeafTask* of the class diagram in Figure 2. The last lines of Listing 2 shows this for the *start()* operation which also represents the *start()* transition in the state charts of Figure 3.

The other state changes behave accordingly to the state chart diagrams in Figure 3 and are expressed in the same way. Almost all transitions can be caused by the user by invoking the operations of the classes shown in Figure 2. Further preconditions for the state changes of tasks are temporal relationships to other tasks in the task model that must not be violated. Thus, there are dependencies and conditions that are not expressed in the state charts of Figure 3.

The only operations that appear in the state chart and cannot be executed by the user are the operations *suspend()* and *resume()*. These are only called implicitly by side effects that are caused by state changes of other tasks in combination with the *SuspendResume* operator. The side effects depending on the temporal relations have to be implemented in the SOIL operations that are part of discussion in subsection 3.4.

The upper diagram of Figure 3 represents the life cycle of a normal task and the lower one of an iterative task. The iterative task can be executed a number of times by calling *start()* and *finish()* operations more than once alternately. In addition, the iterative task can also be directly finished by immediately calling the *finish()* operation. Then the task is executed zero times which is allowed following the CTT semantics. A task has this life cycle if itself or one of its parent's tasks are marked as iterative. This is done by setting the attribute *tempOp* of the class *Task* to the value *ITER*.

3.4 Operational Semantics implemented with SOIL

There are several additional operations used in the metamodel that are implemented with OCL and SOIL and are not shown in Figure 2. The class *Task* for example comprises 60 OCL and SOIL operations. Listing 2 shows the implementation of the *start()* operation from the class *LeafTask*.

Firstly, local variables are declared in the SOIL code of Listing 2 in line 3 and 4. Then they are used in combination with several functions to test if the task is enabled and thus can be started. If the task is suspended by the *OrderIndependency* or *SuspendResume* operator this is not possible. This is tested in the lines 6 and 10.

Further on, it is checked if the task is restricted to be executed by the *Enabling* operator. The predecessor task and its subtasks are not allowed to be in the state *waiting* or *running* if the current task ought to be started.

If the result is negative and the task is enabled to start, the state is set to *running* in line 19. In the following, the side effects depending on the temporal relations are performed in line 20 to 24. Lastly, the pre- and postconditions are specified that ensure the behavior of the task according to its life cycle of Figure 3.

Listing 2. The SOIL implementation for the *start()*-Operation

```
start() begin                                        (1)
 declare
  suspended : Boolean,
  startable : Boolean;
 suspended :=                                        (5)
  self.isSuspendedByOrderIndependency();
 startable := not suspended;
 if (startable) then
  suspended :=
    self.isSuspendedBySuspendResume();               (10)
  startable := not suspended;
 end;
 if (startable) then
  suspended :=
   self.isStartableByEnabling();                     (15)
  startable := not suspended;
 end;
 if (startable) then
  self.state := #running;
  self.skipOptionalTasks();                          (20)
  self.skipChoice();
  self.disableTasks();
  self.suspendTasks();
  self.setStartStamp();
 end                                                 (25)
end
pre leafStart_waiting: self.state=#waiting
post leafStart_running:  self.state=#running
```

4. Workflow Modeling and Execution

In this section we present how workflow models are created and represented in the metamodel-based approach (subsection 4.1). Thereafter, the runtime plugin and a CTT model in execution is introduced in subsection 4.2.

4.1 Workflow Model

Figure 5 shows the abstract syntax for the task tree models that is provided by the USE tool. It shows an UML object diagram that represents the CTT workflow model.

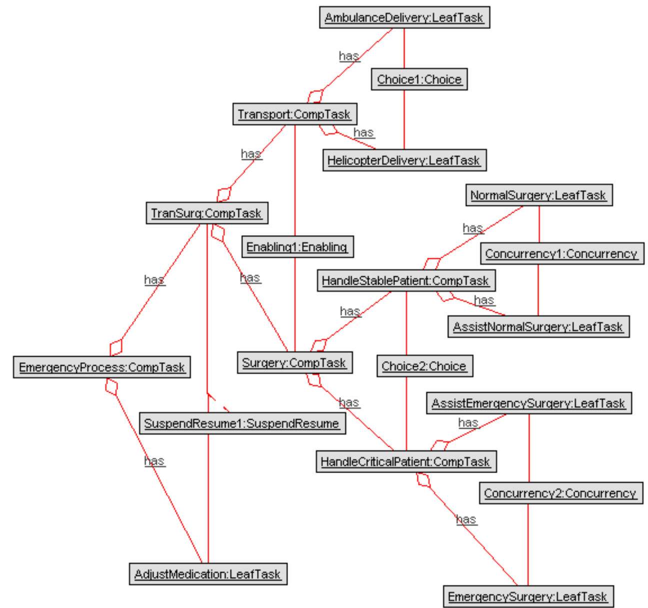

Figure 5: The emergency process of Figure 1 modeled as CTT-diagram with abstract syntax

To provide the object diagram as modeling pane to the user, USE firstly loads the metamodel which is the class diagram of Figure 2 including the OCL- and SOIL specification. Looking at Figure 5, the CTT model in the object diagram is represented more abstractly than the one we have already seen in Figure 1.

Although a concrete syntax is normally better for model understandability, the abstract syntax we use for CTT in Figure 5 has its benefits, too. While the concrete CTT syntax decomposes the tasks downwards, we have chosen decomposition to the right hand side with the abstract syntax. This leads to a better, more compact graphical layout. Thereby bigger process models can be better expressed. This fact can be seen by comparing Figure 1 with 5. Instead of reading the leaf tasks from the left to the right hand side, they have to be read from top to bottom within the object diagram.

In the model of Figure 5 the types of the modeling elements are denoted. The root task *EmergencyProcess* is a composed task that cannot be executed while the leaf tasks starting with AmbulanceDelivery down to AdjustMedication are executable.

4.2 Runtimeplugin for USE

The runtime plugin is implemented for the UML tool USE. It presents the CTT workflow instance to the user also in a tree-like form like shown in Figure 6. The temporal relations of the workflow model are omitted in this presentation. Thus, the model can be displayed in an even more compact form. The execution states of the tasks are presented in specific colors that are listed and related in Table II.

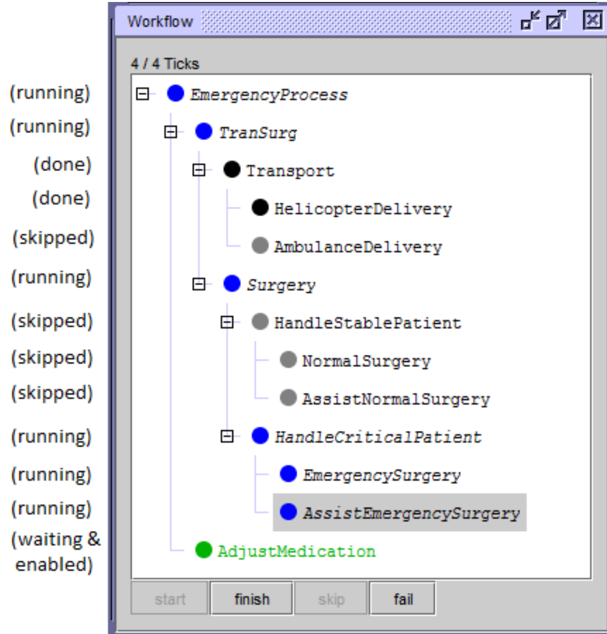

Figure 6: A workflow in execution in the CTT-plugin for the USE tool

We see the task operations that are the ones of *LeafTask* in the metamodel represented to the user by the buttons shown at the bottom of Figure 6. Depending on the ability of the operations to be executed, the buttons are clickable (enabled) or shown greyed out (disabled). An operation is executable if no OCL constraint as pre- or postcondition or invariant is violated. Otherwise that operation is not allowed to be executed. The USE tool tests these properties in advance to show the user which operation is possible to be called and which is disabled. Thus, she is better guided through the workflow execution.

At the current execution snapshot of Figure 6 the task *AssistEmergencySurgery* is selected and in the state running, which is expressed by the blue color (see Table II). Following this state and the upper state chart of Figure 3, the buttons *finish* and *fail* are enabled and *start* and *skip* are disabled.

HelicoperDelivery is marked as black and thus has been executed in the scenario of Figure 6. The task *AmbulanceDelivery* was modeled in a *Choice* relationship to the first task and was consequently *skipped* (marked as grey). *EmergencySurgery* and *AssistEmergencySurgery* have a blue color and are currently running. *HandleStablePatient* and its subtasks were skipped when the task *EmergencySurgery* was started because of the choice relationship to *HandleCriticalPatient*.

AdjustMedication is in the state *waiting* and can be started. The state waiting is marked with the green color and the enabled property is further indicated by the green colored letters. By starting that task the other running tasks would be suspended because of the *SuspendResume* operator in the model of Figure 5.

The whole *EmergencyProcess* is executed when no further leaf tasks are in the state *waiting* or *running* and thus no interaction is possible anymore.

We see in Figure 6 that the composed tasks are also related to states although they have no *state* attribute according to the metamodel of Figure 2. This state is derived from the states of its leaf tasks. It is calculate by an OCL function *getState()* that is part of the metamodel but not deeper introduced here.

TABLE II. COLORS AND THE CORRELATION TO EXECUTION STATES

State	***Color***
Waiting	Green
Waiting (Enabled)	Green (with green letters)
Running	Blue
Done	Black
Skipped	Grey
Failed	Red
Suspended	Turquoise
Undefined	Magenta

5. Related Work

CTTE is the reference implementation for CTT task models to be modeled and simulated in a tool [4]. The developers used an event-based process algebra approach to specify the operational semantics.

Additionally, there are already metamodel-based approaches to describe CTT in [15,16]. They used state chart diagrams to specify task life cycles in [17,18]. An EMF/GMF-based approach implemented as an Eclipse-plugin is presented in [17] for CTT-like models. But in the other metamodel approaches the execution semantics are missing. OCL and SOIL were used for this purpose in the current paper.

In the business process modeling context the Event-driven Process Chain (EPC) modeling language was implemented in [3] with an EMF/GMF metamodel-based approach. The constraint language *Check* was used to check soundness properties of these models.

In [9] it was analyzed that structurdness of workflow models is important for the understandability. In [19] parts of structured programming are analyzed to be used for business process models. *Process Structure Trees* [14] are invented to represent structured workflow models in an analogous way to *Abstract Syntax Trees* in which structured computer programs are represented after program compilation. Jackson Structured Diagrams were invented to design structured programs in a tree model before system implementation [20]. These models are quite similar to CTT task models that similarly produce structured workflow diagrams. CTT is transformed into structured BPMN in [12].

In [8] a task model based approach was used in the context of workflow management systems (WfMS). In that publication the decision modeling aspect was further analyzed for task models. Another WfMS that uses trees for workflow models was

introduced in [13]. With this approach the integration of data and dataflows are part of the models.

For the UML-tool USE there already exist a declarative approach for modeling and executing workflow models [5,6]. In contrast to the declarative more flexible way of modeling workflows, the approach of this paper follows a hierarchical, structured alternative way, where goals can be expressed within the workflow models. The models that are developed with this metamodel promise to be better understood by developers and stakeholders [12] while the approach of [5,6] is more expressive.

6. Conclusion

Although not widely used, the benefits of trees to be applied as workflow models are analyzed and accepted in the business process community. We have put forward the CTT language with a metamodel-based approach. Soundness properties are captured in OCL invariants and are permanently observed by the USE tool. The modeler is indicated at the very moment the problem occurs during design time. In the metamodel the operational semantics are captured by OCL and we used SOIL to implement it.

ASSL was used instead of SOIL to execute the workflow models within the declarative workflow modeling approach [7]. Comparing the ASSL with the SOIL approach from the engineering point of view, the SOIL approach is easier to handle. The SOIL code is directly attached to the operations in the UML class diagram. Thus, only one file is used while with ASSL a second file is needed for the imperative ASSL procedures that are not directly attached to the operations of the UML classes.

To execute these models by the user, we developed a workflow runtime plugin for the USE tool. It uses the SOIL operations of the metamodel. The plugin presents the interface part of the task to the user in an appropriate way. She can interact and validate the workflow model with it. The USE plugin for the declarative approach [7] could be easily adapted for the one presented in this paper. Only the invocation of the operations from the metamodel had to be changed from ASSL to SOIL. The main parts of the user interface remained the same.

We have basically expressed structural soundness properties with OCL invariants in this paper. But to introduce task trees further in the domain of workflow modeling the soundness properties (including operational semantics and deadlocks) have to be further analyzed and captured in the metamodel. Another open issue is the connection to the data model and the decision modeling [6,8]. These aspects are important for workflow models and thus have to be part of the modeling language and metamodel. In the current state, these aspects are expressed outside a metamodel with informal preconditions specifications in CTTE [10].

7. REFERENCES

[1] M. Gogolla, F. Büttner, and M. Richters, “USE: A UML-Based Specification Environment for Validating UML and OCL,” Science of Computer Programming, 69:27-34, 2007.

[2] F. Büttner, and M. Gogolla, “Modular Embedding of the Object Constraint Language into a Programming Language,” 14th Brazilian Symposium on Formal Methods SBMF2011, LNCS vol. 7021, Springer, 2011.

[3] S. Kühne, H. Kern, V. Gruhn, and R. Laue, “Business process modeling with continuous validation,” Journal of Software Maintenance and Evolution: Research and Practice, Volume 22, Issue 6-7, pages 547–566, 2010.

[4] G. Mori, F. Paterno, and C. Santoro, “CTTE: Support for Developing and Analyzing Task Models for Interactive System Design,” IEEE Transactions on Software Engineering, 2002, pp.797-813.

[5] J. Brüning, M. Gogolla, and P. Forbrig, “Modeling and formally checking workflow properties using UML and OCL,” 9th International Conference BIR2010, LNBIP vol. 64, Springer, 2010.

[6] J. Brüning and M. Gogolla, “Metamodel-based Workflow Modeling and Execution,” 15th International Enterprise Distributed Object Computing Conference (EDOC2011), IEEE, 2011.

[7] J. Brüning, L. Hamann, and A. Wolff, “Extending ASSL: Making UML Metamodel-based Workflows Executable,” 11th International Workshop OCL2011, ECEASST vol. 56, 2011.

[8] J. Brüning, P. Forbrig, “TTMS: A Task Tree Based Workflow Management System,” 12th International Conference BPMDS2011, LNBIP vol. 81, Springer, 2011.

[9] R. Laue, and J. Mendling, “Structuredness and its significance for correctness of process models,” Inf. Syst. E-Business Management 8(3): 287-307, 2010.

[10] F. Paterno, “Model-Based Design and Evaluation of Interactive Applications,” Springer, 2000.

[11] B. Meyer, “Applying "Design by Contract",” IEEE Computer 10(25):40–51, 1992.

[12] J. Kolb, M. Reichert, and B. Weber “Using Concurrent Task Trees for Stakeholder-centered Modeling and Visualization of Business Processes,” 4th International Conference S-BPM ONE 2012, CCIS vol. 284, Springer, 2012.

[13] M. Weske, “Workflow Management Systems: Formal Foundation, Conceptual Design, Implementation Aspects,” Postdoctoral Dissertation, University of Münster, 2000.

[14] J. Vanhatalo, H. Völzer, and J. Koehler, “The Refined Process Structure Tree,” 6th International Conference BPM2008, LNCS vol. 5240, Springer, 2008.

[15] R. Bastide, and S. Basnyat, “Error Patterns: Systematic Investigation of Deviations in Task Models,” 5th International Workshop TAMODIA2006, LNCS vol. 4385, Springer, 2006.

[16] R. Bastide, “An Integration of Task and Use-Case Meta-models,” 13th International Conference HCI2009, LNCS vol. 5610, Springer.

[17] D. Reichart, and P. Forbrig, “Transactions in Task Models,” 7th International Workshop TAMODIA2008, LNCS vol. 5247, Springer, 2008.

[18] B. Bomsdorf, “The WebTaskModel Approach to Web Process Modelling,” 6th International Workshop TAMODIA2007, LNCS vol. 4849, Springer, 2007

[19] V. Gruhn, and R. Laue, “What business process modelers can learn from programmers,” Sci. Comput. Program. 65(1): 4-13, 2007.

[20] J. R. Cameron, "JSP & JSD: The Jackson approach to software development," IEEE Computer Society Press, 1983.

[21] A.-W. Scheer, “ARIS: Business Process Modeling,“ Springer, 2000.