Formalizing UML-based Process Models Using Graph Transformations *

Ansgar Schleicher

Aachen University of Technology
Department of Computer Science III
D-52056 Aachen, Germany
schleich@i3.informatik.rwth-aachen.de

Abstract. Supporting technical development processes through process management environments is vital for a project's success. While process enactment enables a project manager to plan and monitor a process and guides the participating developers, process modeling aims at understanding, communicating and reusing process descriptions. Thus, requirements for languages supporting process enactment are quite different from those for languages supporting process modeling.
In this paper we demonstrate how the task of process modeling can be tackled using a standard object-oriented modeling notation, the Unified Modeling Language. By transforming the resulting model into the formal notation of an underlying generic process model, we support its enactment. This generic model has been formally specified within the graph transformation system PROGRES.
In this way we are able to provide suitable languages for process modeling and enactment within one coherent environment.
Keywords: Process Modeling, Process Enactment, Graph Transformations, Unified Modeling Language

1 Introduction

The success of technical development processes is dependent on the coordination of the participating developers [19]. A process management environment supports coordination by providing tools to plan, enact and monitor a process model and by providing developers with work contexts and information about their responsibilities, deadlines etc.

To support development processes adequately, a management environment is required to be able to deal with their inherent dynamism. This dynamism arises through changing requirements, feedback to earlier stages of the development process, simultaneous and forward engineering, moved deadlines and shrinking budgets. The architecture and functionality of a suitable management environment has been presented in [18] and [6].

* The work described in this paper was partially supported by the Deutsche Forschungsgemeinschaft within the collaborative research center 476 ("IMPROVE"). Within this project we cooperate with Bayer AG, Leverkusen, Germany.

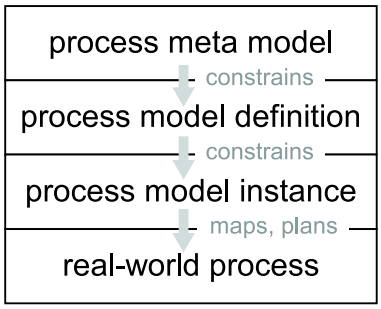


Fig. 1. Modeling framework

This paper will focus on adequate modeling support within such a management environment. In the context of development processes the modeling framework of Figure 1 is widely spread. It distinguishes four levels. A process meta model provides a process modeling language for process model definitions. The latter are schematic, type-level process models and valid instances of the process meta model. Process model instances are valid instances of a process model definition and are mappings of the real-world process. With respect to development processes, process model definitions are the means to define reusable process specific knowledge, since the instance level is highly dynamic and resulting structures cannot be reused in most cases.

Within the AHEAD [1] environment *dynamic task nets* have been developed as a process meta model [4, 5]. The meta model supports the continous structural evolution during process enactment and thus meets the formulated requirements (cf. section 2). Dynamic task nets have been formally specified in the Programmed Graph Rewriting System (PROGRES) [13].

The question arises how domain specific schematic process knowledge is fed into the management environment to restrict the structure and behavior of the instance level task nets. Extending the PROGRES specification of dynamic task nets is a very unattractive approach, since it requires an expert on graph transformations as a process modeler. Additionally, experience has shown that processes cannot be modeled very intuitively.

In contrast, an object-oriented modeling approach is very attractive, since dynamic task nets are evolving structures of interrelated and interacting objects. As an object-oriented modeling language the *Unified Modeling Language* (UML) [1] is the natural choice. It bears the implicit advantages that process model definitions can be communicated easily to a large number of people and that it contains a number of diagrams that are very appropriate for structural and behavioral process modeling on the type level.

However, in providing abstract UML-based process model definitions, we lose the ability to enact them. This paper will describe how we can make the two

[1] Adaptable Human-Centered Environment for the Administration of Development Processes

ends meet. It will show how the UML can be applied to create process model definitions (cf. section 3) and it will describe how these models can be formalized by transformation into an extension of the graph grammar based specification of dynamic task nets which then contains the meta model and a valid generated process model definition.

2 Dynamic Task Nets

As a formal process meta model, dynamic task nets have been introduced, which provide mechanisms to model development processes and their inherent dynamics. Modeling a process as a dynamic task net means splitting the overall process into subprocesses and to clarify their respective behavior. A task consists of the description of what is to be done, the *task interface*, and of a description of how it is to be done, the *task realization*. Dynamic task nets may contain complex (refined) and atomic realizations.

Tasks are connected through task relations. *Control flow* relations introduce a temporal ordering on tasks. *Feedback flow* relations are always directed oppositely to control flow relations and are inserted to mark iteration or exception steps of a process.

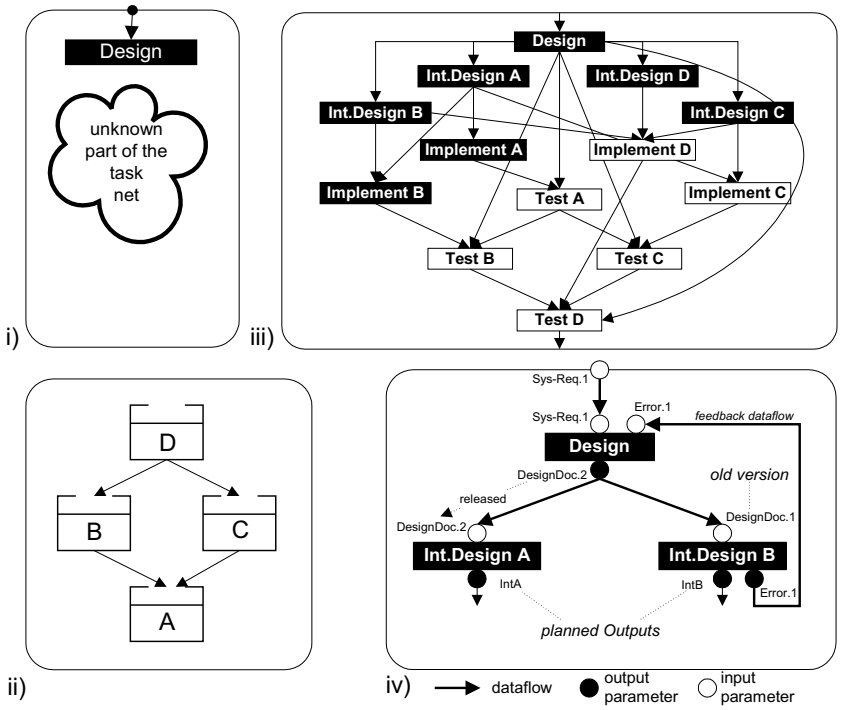


Fig. 2. Example of a dynamic task net

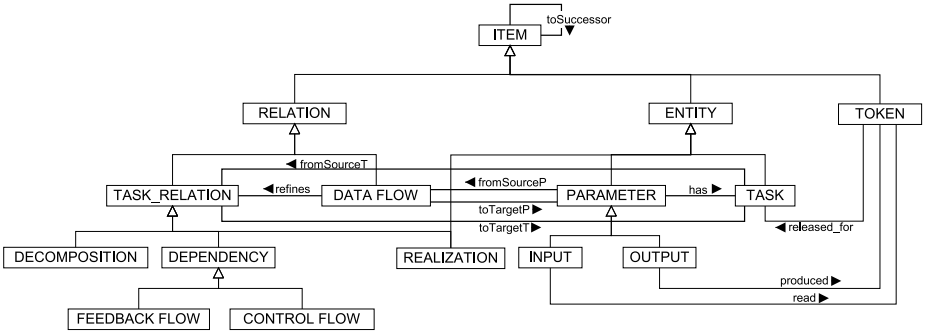


Fig. 3. Graph schema for dynamic task nets

Outputs to be produced and *inputs* needed by a task are modeled as parameters of a task's interface. Between parameters *data flow* relations may be introduced which indicate the flow of products through a task net

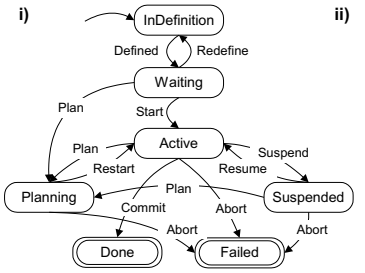
Figure 2 gives a small example of a dynamic task net's structure [2] and its evolution during enactment. At the beginning of a software project only little is known about the development process. A design task is introduced into the net (part i), while the following structure remains unspecified as it depends on the design document's internal structure (part ii). As soon as this is produced, the task net can be completed (part iii).

Output documents of tasks are versioned and can be released on a task-by-task basis. Part iv) shows how a version of a design document is produced after feedback occurred from the task implementing module B to the design task. This new version is at first only selectively released.

Internally dynamic task nets are represented as graphs of typed nodes and edges. A *task graph* as the internal data structure of a process model instance consists of nodes representing tasks, parameters and tokens as references to products created during the process. Task relations and data flows are internally represented by nodes, because they carry attributes and neither PROGRES nor the underlying graph database support attributed edges. Edges are used to connect task, parameter and token nodes.

In order to restrict the graph to meaningful structures with respect to the process meta model, a graph type is defined by a *graph schema*. The process meta model's graph schema is displayed in Figure 3. Node class `ITEM` serves as the root class. On the next layer of the inheritance hierachy we mainly distinguish between process entity and process relationship types. The node class `TOKEN` describes nodes representing tokens that are passed along data flows. `TASK` nodes own `PARAMETER` nodes which are either `INPUT` or `OUTPUT` parameters. Tasks can `produce` tokens via output parameters and `read` tokens via input parameters. Tasks have an assigned `REALIZATION` which determines a subnet's structure and serves as a vertical `TASK RELATION`. Horizontal relationships are `CONTROL FLOW`

[2] In this case an instance of a standard unconstrained process model definition is used to demonstrate the concept of dynamic task nets



ii)

```
transaction STD_AbortTrans( Task : STD_TASK) =
     ((Task.State in ("Active" or "Suspended" or "Planning"))
      and ((Task.=ToParent=> : STD_TASK.State = "Active")
           or Task.IsRoot                                  )   )
  & for all t : STD_TASK := Task.=CurrentChildren=> : STD_TASK do
      choose
        when (t.State in ("Active" or "Planning" or "Suspended"))
        then
          STD_AbortTrans ( t )
      else
        skip
      end
    end
  & Task.State := "Failed"
end;
```

Fig. 4. State transition diagram and its formal specification

or `FEEDBACK FLOW` relations. Parameters in turn are connected via `DATA FLOW` relations.

To consistently manipulate and enact a task net, a set of operations for editing and execution are specified as *graph transformations* and procedures of these. Task nets can be edited by introducing new entities and relationships into the net or by removing some. Operations to execute a task net deal e.g. with the change of task states and the token game between tasks. Please note that editing and execution of task nets are both described using a uniform mechanism. This allows us to specify the intertwined editing and execution of task nets [3].

The executional semantics of dynamic task nets are based on *cooperating state machines*. Every task has a lifecycle determined by the state transition diagram displayed in part i) of Figure 4. The state diagram is formally specified in PROGRES by supplying an attribute `State` for the node class `TASK` and by specifying a set of transactions, one for each state transition.

Some standard behavior of dynamic task nets is defined with respect to this state transition diagram.

- State transitions may influence the context of the task performing the state change. Abortion of a task leads to the abortion of all of its subtasks (cf. part ii), Figure 4).
- All editing and execution operations are dependent on the current state of the task to be manipulated or its parent task. Operations for editing, like insertion of a new dataflow, can only be executed if the parent task of the subnet in question is currently in states `InDefinition` or `Planning`, while operations for performing the token game can only operate on tasks being in state `Active`.
- Tasks can only be activated if all predecessors left state `Waiting` and can only be committed if all predecessors have been successfully completed.

In order to allow for the handling of dynamic, non-anticipated situations in the model, every state transition and every operation is followed by an event (cf. `SendAbort`-statement in Figure 4, part ii).

[3] For a detailed description on the use of graph transformations within the formal specification of dynamic task nets see [4]

We presented the main features of a generic process model. In order to manage a certain process, domain-specific knowledge has to be provided. A domain-specific process structure introduces new task and parameter types and refines the generic relationship types. Process behavior has to be defined in terms of executional constraints in relation to the generic state diagram and specific event-handlers. The following section will show how this adaptation of the generic process model can be accomplished.

3 Using UML for Process Model Definition

Modeling domain-specific development processes based on dynamic task nets by extending the meta model's specification is a rather tedious task [10]. Especially for someone who is not an expert on graph transformations but an expert in the technical development domain, a considerable abstraction from the underlying mechanism has to be found. While structural model definition can at least take place in a straightforward manner by extending the graph schema, there is no intuitive mapping between the concepts of behavioral model definition and PROGRES' language constructs. For this reason it is necessary to provide a modeling tool within the AHEAD-environment that allows a non-expert on graph transformations to model a process. To this end we use the Unified Modeling Language (UML).

3.1 Defining a UML Extension to map the Process Meta Model

The UML is not designed in conformance with our process meta model. Thus, we would like to restrict the UML's language constructs to meaningful ones with respect to our application domain. However, the UML does not provide a full-fledged meta modeling facility. Defining a meta model can only be achieved by extending the UML's own meta model, which would result in a UML-variant. The only way to specify a meta model in conformance with the UML is to define *stereotypes* and *constraints* on these. A stereotype is a virtual meta class refining one of UML's own meta classes and thus enables a modeler to give the standard modeling elements additional semantics. Providing constraints leads to a restriction of possible diagram structures. Using stereotypes for meta modeling is not very satisfactory, since many syntactical and semantical constraints cannot be expressed. However, this way of "'meta modeling"' is conformant with the OMG's proposal to define UML extensions [15]. A cutout of the extension used for our modeling approach is described in this section.

Figure 5 i) displays a table with the used stereotypes and the symbols representing them in UML diagrams. UML's meta class "class" is refined by meta classes for task, input and output parameter and realization classes. In addition UML's meta class "package" is refined to categorize packages related to their content. Various stereotypes are established to distinguish between different associations. The constraints that have to hold on these association subclasses are defined in a separate table in Figure 5 ii).

i)

UML meta class	Stereotype	Symbol
class	Task	
class	Input	
class	Output	
class	Realization	
package	...	
association	cflow	
association	fback	
association	dflow	
association	may_contain	
association	may_have	
association	may_realize	

ii)

From \ *To*	Task	Input	Output	Realization
Task	cflow, fback	may_have	may_have	
Input		dflow		
Output		dflow	dflow	
Realization	may_realize, may_contain			

Fig. 5. Stereotypes of the UML extension

The table defines the source and target types of associations carrying the given stereotype. Additional context sensitive constraints can be defined using the *Object Constraint Language* (OCL) [17] wich is part of the UML. Since understanding these constraints requires a detailed knowledge of the UML meta model they are omitted here.

3.2 Structural Process Model Definition

In our approach the UML is used for process model definition. Resulting *process model definitions* thus abstract from a multitude of instance level task nets (an example of which was presented in Section 2). A process model definition's structure consists of task, parameter and realization classes and their various interdependencies. Consequently, we use UML's *class diagrams* to model this structure.

Conceptually, we distinguish a task's *interface* from its *realization*. The interface defines a task's contract such as its parameter profile and its external behavior. It abstracts from possible realizations, one of which can be selected by the corresponding actor — e.g. a project manager — at enactment time.

Figure 6 shows the interface of the task class Develop Software System at the top. The shown interface consists of the task class and its composed input and output parameter classes. Cardinalities may be defined together with the compositions to restrict the number of parameters of a certain class on the instance level. The interface is stored in a separate package[4].

A task class composes all its respective realization classes (symbolized by a doubly rimmed rectangle). Since the interface abstracts from a set of possible realizations, the realization classes are not part of the interface of a task package. Rather an individual package is introduced for every realization class. Realization

[4] The use of the UML's package concept to structure a process model and to encourage reuse of model fragments will not be explained in this paper. A detailed explanation can be found in [7].

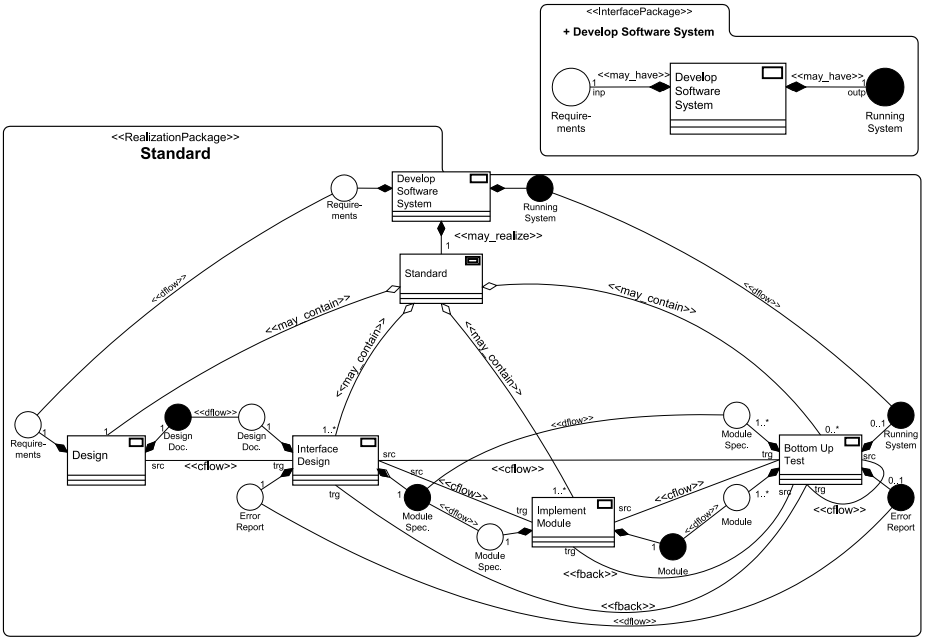


Fig. 6. Task interface and realization

classes abstract from complex subprocess definitions as shown in Figure 6. These allow for the *composition* of other task classes through control flow (stereotype ≪cflow≫) and feedback flow (stereotype ≪fback≫) associations. The direction of these associations is indicated through the rolenames src and trg. For example, the Interface Design and Implement Module task classes are connected by a control flow association which indicates that implementation of modules cannot take place before work on a corresponding module specification was started (however, execution may overlap). Analogously, feedback is allowed between the Bottom-Up Test and the Interface Design task classes in case of errors during the test. Of course, feedback might as well occur in other parts of this subprocess model.

Control flow and feedback flow associations define potential channels for data flow, which is explicitly modeled through *data flow* associations (stereotype ≪dflow≫) between parameter classes. Data flow associations can be introduced between an input and an output parameter class, with the output class playing the role of source. Vertical data flow can be defined between two input or two output parameters. This gives the complex parent task the possibility to supply the refining tasks with their input data and to receive their results.

In our example the Design task is supplied with the initial requirements document. There, the requirements are analyzed and a design document is created, which is subsequently sent to the Interface Design tasks. After all modules have been implemented and successfully tested, the running system is sent to the parent task as the result of process execution.

Again, cardinalities are used to restrict the number of instances of the classes at enactment time. In the example the number of Design tasks is restricted to exactly one, while Interface Design and Implement Module tasks may occur in any number, which depends on the number of modules to be implemented.

3.3 Behavioral Process Model Definition

After we have shown how a process model definition can be structurally specified with class diagrams (and packages) we will now turn our attention to behavioral modeling.

One way to model a task class' specific behavior is to define *conditions* for the transitions of the state diagram. These are formulated in the OCL and provide the means to influence the way a task net is executed. Different development policies like concurrent and sequential engineering can be realized.

A cutout of a sample state diagram for the Bottom-up Test task class is shown in Figure 7. It contains a condition for the start transition, which may be executed only when all inputs are available (the standard behavior does not demand this).

Since the presence of all inputs of the Bottom-up Test if enforced, we can now automate their consumption. The UML allows for the definition of actions inside of a state to automate execution steps. An action can be triggered by entering or by leaving a state. The sample state diagram of Figure 7 leads to the automatic consumption of all inputs when the state Active is entered.

Every method being executed by a task by default sends out a corresponding event to its predecessors, successors, children, and parent. The underlying process engine provides an *event/trigger mechanism* that triggers the execution of event handlers in the receiving objects. These event handlers enable task objects to react to actions performed in their individual context.

Figure 7 shows how the set of event-receiving tasks can be restricted through send-clauses at specific transitions. The sample state diagram shown in Figure 7 restricts the target set of the CreateFeedback event to the task's parent.

Defining the specific behavior of a process model includes the specification of custom event handlers. An *event handler* can be specified for every task class. The UML allows to specify a method's semantics in any language, like C++ or pseudo code. For example, the automatic activation of a task, dependent on

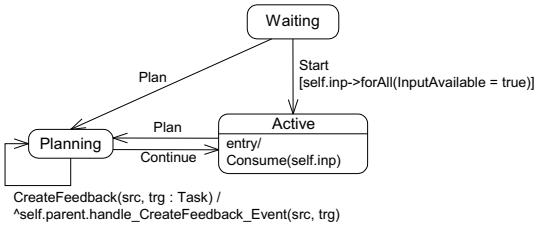


Fig. 7. Cut-out of a specific state diagram

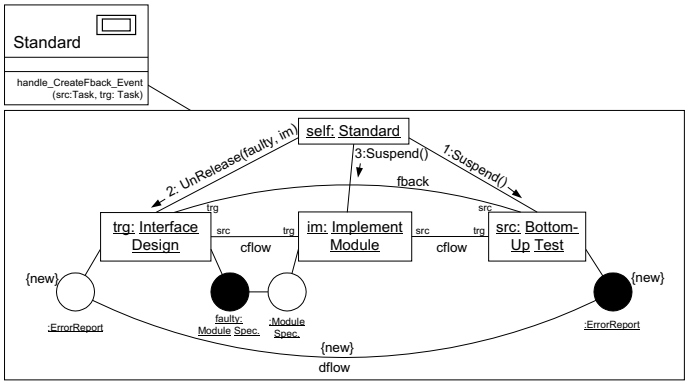


Fig. 8. Complex event handler

certain data being released by a predecessor, can be modeled with task class specific event handlers. However, the expressiveness of these event handlers is very limited since the task class' context is not known at the time of its specification (in fact, a task class may be reused in different contexts, i.e. realizations of complex tasks). We therefore allow for the definition of realization class specific event handlers. A realization receives all events that are being sent to its assigned task. By this way events being sent to the parent of a task net can lead to very complex *task net transformations* since the structure of the subprocess is well known to the realization. If processes are enacted multiple times, process knowledge grows and many matters can be handled automatically.

Complex event handlers can be specified through UML's *collaboration diagrams* which are used to define the semantics of an event handling method. A collaboration diagram consists of objects and links that are required to be available when the method is executed. Additionally, objects and links can be created and destroyed during the execution of the specified method. The communication between objects can be defined through messages which may refine links.

Figure 8 gives an example of how a CreateFeedback event can be handled automatically through this mechanism. The event handler is defined for feedback occurring between tasks of type Bottom Up Test and Interface Design. In addition to the feedback flow's source and target objects we search for an intermediate task of type Implement Module and some parameter objects. The event handling method then creates two parameter objects that allow exchange of an error report. Objects marked with the constraint {new} will be created during the execution of the event handler. Destruction of objects can be achieved through the constraint {destroy}. By installing a data flow link between these parameter objects the feedback flow is refined and an error report can be sent to the feedback flow's target. *Links* created by the method are marked with the constraint {new} as well. Finally, the tasks' behavior is specified through *messages*. In the example case, the feedback flow's source is suspended (message 1), the erroneous output document of the feedback flow's target retracted (message 2) and the intermediate implementation task suspended (message 3).

4 Formalizing UML-based Process Model Definitions

Enacting a modeled process requires the generic model to consider the domain-specific structural and behavioral constraints. In terms of the process structure it means that only modeled classes can be instantiated and related according to the process model definitions in consideration of the modeled cardinalities. In terms of process behavior it means that conditions from refined state transition diagrams have to be checked and automated actions have to be invoked. Modeled event-handlers have to be triggered and successfully completed if their pre-condition holds during enactment. To reach these aims, the UML model is transformed into an extension of the meta model's PROGRES specification (cf. section 2), thus providing a formalized interpretable version of the process model definition.

4.1 Transforming the Structural Model Definition

The structural model definition can be transformed into PROGRES code by extending the meta model's graph schema with new node types. Each class marked with the stereotypes task, realization, input or output is transformed into a node type instantiating the corresponding node class. Each association marked with a stereotype cflow, fback or dflow is transformed into a node type instantiating the corresponding node class in the same fashion. Meta attributes (which are schema-level attributes) are used to fix the relations between these node types and to constrain the cardinalities.

In the following we will present examples from the transformation of the software development process model definition introduced in figure 6.

A task class is transformed into a node type as an instance of node class `TASK` (cf. Figure 9). The inherited meta attributes of the node class are refined for the node type to reflect the associations a task class has to other classes in the UML model. In particular the attribute `DeclaredRealizations` fixes the set of realization types corresponding to the task's interface and thus reflects

```
node class TASK is a ENTITY
  meta
    DeclaredParameters : type in PARAMETER [0:n] := PARAMETER;
    OblParameters : type in PARAMETER[0:n] := nil;
    MultParameters : type in PARAMETER[0:n] := PARAMETER;
    DeclaredRealizations : type in REALIZATION [0:n] := REALIZATION;
end;

node type Bottom_Up_Test : TASK
  redef meta
    DeclaredParameters : Module or Module_Spec or Running_System or
                         Error_Report;
    OblParameters : Module_Spec or Module;
    MultParameters : Module_Spec or Module;
    DeclaredRealizations : WhiteBoxTest or BlackBoxTest;
end;
```

Fig. 9. Transformation of task classes

```
node type Standard : REALIZATION
  redef meta
    DeclaredInterface : Develop_Software_System

    ChildTasks : Design or Interface_Design or
                 Implement_Module or Bottom_Up_Test;
    OblChildTasks : Design or Interface_Design or Implement_Module;
    MultChildTasks : Interface_Design or Implement_Module
                     or Bottom_Up_Test

    TaskRelations : CF_Des_to_IntDes or FB_Test_to_IntDes or ...
    ParameterRelations : DF_MSpec_to_MSpec or ...
end;
```

Fig. 10. Transformation of realization classes

associations of stereotype may_realize. In the same fashion the set of parameter types of this interface is constrained and thus associations of stereotype may_have are reflected. The difference is that in this case cardinalities have to be taken into account. For this reason three meta attribute are used to fix the set of all possible parameter types, the ones with obligate cardinality (1, 1..*) in the UML model and the set of parameter types with multiple cardinality (0..*, 1..*).

Transforming a realization class is very similar. A new node type is introduced as an instance of the node class `REALIZATION` and the meta attributes are redefined to express the structural constraints defined in the UML model. Figure 10 shows the schema extension performed for the realization class Standard from Figure 6. In this case meta attributes are used to reflect the associations of stereotype may_contain and their cardinalities. In addition the sets of possible task and parameter relation types are stored in meta attributes.

Task and parameter relations themselves are mapped into separate node types. In this case meta attributes are used to constrain the source and target types of the relation and to define their respective cardinalities.

When an enacted process model instance is supposed to be structurally manipulated, the corresponding graph transformations check the meta attributes to enforce *structural consistency* with the process model definition. Within the graph transformation for e.g. creation of a subtask it is checked whether the type of the task to be created appears within the `ChildTasks` attribute of the supertask's assigned realization.

4.2 Transforming the Behavioral Model Definition

The generic state transition diagram can be enhanced by project specific transition conditions and automated action calls. As we have shown before, each transition or generic operation is internally specified as a PROGRES transaction. The problem that arises when transforming the behavioral model is that we have to map an object-oriented model to a non-object-oriented specification language. The generic operations are not directly related to a node class. Therefore, they cannot be redefined for instances of a node class in the same manner as meta attributes can.

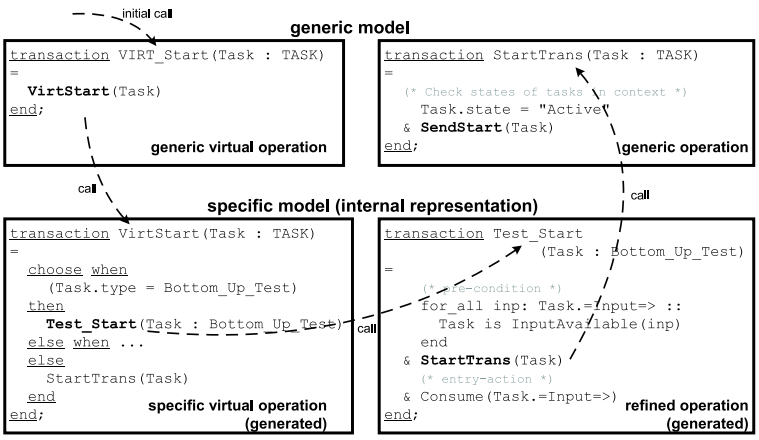


Fig. 11. Transformation of a refined operation

For this reason, we have to simulate the binding of operations to objects in the specification. We introduce a new transaction which simulates the *late binding* of operations. It receives a node as an actual parameter and determines the operation to call from the node's type.

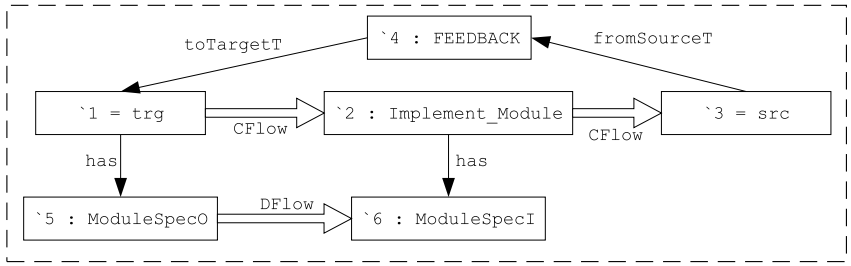
Inside of a transaction reflecting a refined transition the exit-actions of a transition's source state can be invoked, the transition's pre-condition can be checked, the transition fired (by calling the generic transition)[5] and finally the entry-actions of the new state can be invoked. For all other generic operations (those that do not trigger a state change) the exit- and entry-actions of the source and target states are omitted, because no state change is performed.

Figure 11 shows how the `Start` transition modeled in Figure 7 is transformed into a PROGRES transaction and how the flow of control simulates late binding. Within the meta model a transaction `VIRT_Start` is available which calls a generated transaction `VirtStart`. Within `VirtStart` an entry exists for every task type that refines the generic start transition. Within that entry the refined transition (e.g. `Test_Start`) is called, where conditions are checked and automated actions invoked. Within the refined transition (operation) the call to the generic transition (operation) is placed.

Graphical definitions of event handlers as presented in Figure 8 are transformed as follows: Objects and links that are required to be available for the event handler's execution (i.e. a feedback's source and target) are transformed into a PROGRES *graph query*. A graph query searches for and returns the graph nodes representing the needed process objects. Objects and links created during the method's execution are created within the graph through the specified meta model's operations of the generic model. All message calls to objects (i.e.

[5] It is very important to call the generic transition as it defines the standard behavior of dynamic task nets and performs the state change. The call has to be placed between the source state's exit-actions and the target state's entry-actions

```
test Thandle_CreateFeedback_Event( src : Bottom_Up_Test;
                                   trg : Interface_Design;
                                   out FB : FEEDBACK;
                                   out impl_Module : Implement_Module;
                                   out mspec1 : ModuleSpecO;
                                   out mspec2 : ModuleSpecI; )
=
```




```
return FB := `4; impl_module := `2;
       mpspec1 := `5 ; mspec2 :=`6;

transaction handle_CreateFeedback_Event( src:Bottom_Up_Test,
                                         trg:Interface_Design) =
  (* declaration of local variables for process objects*)
    Thandle_CreateFeedback_Event(...)
    CreateParameter(src, Error_ReportO, out er1)
  & CreateParameter(trg, Error_ReportI, out er2)
  & CreateDataflow(er1, er2, FB)
  & Suspend(src)
  & UnRelease(mspec1, impl_Module)
  & Suspend(impl_Module)
end;
```

Fig. 12. Transformed event handler

messages 1-3 in Figure 8) are transformed into corresponding calls to the meta model's operations.

The searching of an object structure and implanting of an enhancement to this object structure could best be realized as a graph transformation within PROGRES. However, the creation and deletion of process objects through a graph transformation would ignore the semantics of dynamic task nets which are contained in the base operations.

Figure 12 shows a sample graph query and transaction for the collaboration diagram in Figure 8. Starting with the node for representing the created feedback flow, nodes representing the feedback flow's source and target task, the affected tasks and significant parameters are extracted from the graph and returned to the transaction which calls the appropriate base operations on these objects.

5 Related Work

Our approach combines the standard object-oriented modeling language UML with graph transformation. To our knowledge this approach is unique with respect to process management environments except for the predecessor of this

approach, called MADAM[6]. MADAM introduced an own set of (visual) languages for process model definition as an abstraction of PROGRES. However, we realized that many of MADAM's features could be mapped into diagrams of the UML. Using the latter gives us the benefit of easier communication of models to others. Additionally, we were able to further raise the level of abstraction.

However, there exist other approaches that transform high-level process models to an underlying formal process programming language, such as ESCAPE [8]. ESCAPE is based on OMT which is one of the predecessors of the UML. It consists of an object-model (EER-diagram) to model a product structure on the type level, a coordination model (statecharts) for behavioral modeling and an organization model (tables). These models are transformed into rules of the rule-based process programming language MERLIN [11], which then executes the rules through forward and backward chaining.

Besides these approaches based on an indirect execution paradigm there exist approaches supporting direct execution like those based on procedural programming [14] or Petri Nets [2]. A process is modeled by programming in the directly executable languages which are usually on a very detailed level. Abstraction from the paradigm used and visual modeling are not supported.

Other publications introduce ways to model business processes with UML using activity diagrams [9, 16]. While activity diagrams have been invented to model workflows, they are very inadequate for modeling development processes as these are hard to predict and evolve continously. Modeling development processes in class diagrams is more adequate since processes consist of dynamically changing configurations of interacting objects.

With FUJABA [3] there exists an approach that allows to model graph transformations, called story patterns, and graph schemas in UML. Control flow is specified using activity diagrams. Specifications are executable but FUJABA is still not a suitable approach for software process modeling: Having specified a generic model using FUJABA, domain-specific extensions would have to be specified as schema extensions for the structural model (where inheritance of associations is unwanted) and activity diagrams for the behavioral model. However, within an activity diagram e.g. a complex task net transformation could not simply be modeled as a story pattern because data abstraction would be violated. Rather methods of various objects would be called. This approach is the equivalent to defining a specific process model in PROGRES itself and strongly counteracts our aim to use UML for software process modeling but still offer a process modeler language constructs suitable to the modeling domain.

6 Conclusion and Future Work

In this paper we presented dynamic task nets and their formal graph transformation based specification. Dynamic task nets were developed to support a process manager in guiding and monitoring development processes and to support technical developers in coordinating their work. Customizing dynamic task nets to

[6] Management and Adaptation of Administrational Models [12, 10]

a domain-specific process and providing process knowledge can be achieved by using the UML. Enaction of dynamic task nets takes the modeled structural and behavioral constraints into account, because the UML model is transformed into an extension of their formal specification. The graph schema is extended with specific types and the generic operations can be refined type-specifically.

Benefits of this approach are that we can offer a high-level language for defining process models without losing the ability to enact them. Using the UML further simplifies process modeling as it is widely spread. Using graph transformations for the specification of a the syntax and semantics of a process meta model on the other hand, enables us to formalize execution and manipulation of task nets with a uniform mechanism. The occuring gap between UML models and enaction support is closed by providing a mapping between informal (UML) and formal (PROGRES) process model definitions as described in section 4.

A process can be modeled comfortably using the commercial CASE tool Rose from Rational. After the model has been transformed into a PROGRES specification, PROGRES' code generation mechanism and the associated framework for graph based applications can be used to build a domain-specific process management environment. As a proof of concept, the transformation has been implemented as an extension to Rose.

Experience shows that model definition and enaction of instances cannot necessarily take place sequentially. Rather, a process model definition has to be adapted, because process requirements changed, process deadlines have passed etc. For this reason the environment has to support evolution of the process model definition. This allows a process modeler to change the model definition at enactment time and propagate these changes to the instance-level on demand. The state of process execution has to be preserved during the triggered task net reorganization. Problems arise through the incapability of the PROGRES environment to deal with graph schema changes, but conceptual solutions have been found to overcome this deficit.

References

1. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison Wesley, 1998.
2. W. Deiters and V. Gruhn. The FUNSOFT net approach to software process management. *International Journal of Software Engineering and Knowledge Engineering*, 4(2):229–256, 1994.
3. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph grammar language based on the unified modelling language and java. In G. Engels and G. Rozenberg, editors, *Proceedings TAGT'98 – 6th International Workshop on Theory and Applications of Graph Transformation*, LNCS 1764. Springer-Verlag, Heidelberg, Germany, 2000.
4. P. Heimann, G. Joeris, C.-A. Krapp, and B. Westfechtel. DYNAMITE: Dynamic task nets for software process management. In *Proc. ICSE '18*, pages 331–341, Berlin, Mar. 1996.

5. P. Heimann, C.-A. Krapp, and B. Westfechtel. An environment for managing software development processes. In *Proc. 8th Conf. on Software Engineering Environments*, pages 101–109, Cottbus, Germany, Apr. 1997.
6. D. Jäger, A. Schleicher, and B. Westfechtel. AHEAD: A graph-based system for modeling and managing development processes. In *Proceedings AGTIVE'99*, this volume.
7. D. Jäger, A. Schleicher, and B. Westfechtel. Using UML for Software Process Modeling. In O. Nierstrasz and M. Lemoine, editors, *Proc. ESEC/FSE '99*, LNCS 1687, pages 91–108, Heidelberg, 1999. Springer-Verlag.
8. G. Junkermann. A dedicated design language based on EER-models, statecharts and tables. In *Proc. 7th International Conference on Software Engineering and Knowledge Engineering*, pages 487–495. Knowledge Systems Institute, 1995.
9. A. Korthaus. Using UML for business object based systems modeling. In M. Schader and A. Korthaus, editors, *The Unified Modeling Language — Technical Aspects and Applications*, pages 220–237. Physica-Verlag, Heidelberg, Germany, 1998.
10. C.-A. Krapp. *An Adaptable Environment for the Management of Development Processes.* Number 22 in Aachener Beiträge zur Informatik. Augustinus Buchhandlung, Aachen, Germany, 1998.
11. B. Peuschel and W. Schäfer. Concepts and implementation of a rule-based process engine. In *Proc. 14th International Conference on Software Engineering*, pages 262–279, 1992.
12. A. Schleicher. High-level modeling of development processes. In B. Scholz-Reiter, H.-D. Stahlmann, and A. Nethe, editors, *Process Modelling*, pages 57–73, Cottbus, Germany, Feb. 1999. Springer-Verlag.
13. A. Schürr, A. Winter, and A. Zündorf. Graph grammar engineering with PROGRES. In *Proc. ESEC '95*, LNCS 989, pages 219–234, Barcelona, Spain, Sept. 1995.
14. S. M. Sutton, D. Heimbigner, and L. J. Osterweil. APPL/A: A language for software process programming. *ACM Transactions on Software Engineering and Methodology*, 4(3):221–286, July 1995.
15. UML Revision Task Force. *OMG UML Specification v. 1.3 beta R1 draft.* Object Management Group (OMG), http://uml.systemhouse.mci.com, 1999.
16. G. Versteegen. Objektorientierte Geschäftsprozeßmodellierung mit der UML: die Innovator Business Workbench. *OBJEKTspektrum*, (1):62–67, Jan. 1998.
17. J. Warmer and A. Kleppe. *The Object Constraint Language - Precise Modeling with UML.* Object Technology Series. Addison-Wesley, 1999.
18. B. Westfechtel. *Models and Tools for Managing Development Processes.* LNCS 1646. Springer-Verlag, Heidelberg, Germany, 1999.
19. S. Zahran. *Software Process Improvement - Practical Guidelines for Business Success.* Addison-Wesley, 1997.