Using UML for Software Process Modeling

Dirk Jäger, Ansgar Schleicher, and Bernhard Westfechtel

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

Abstract. We examine the benefits of using an object-oriented modeling language for software process modeling. We show how the Unified Modeling Language (UML) can be used to model software processes based on dynamic task nets, which evolve continuously during enactment. We have selected UML for various reasons: it is wide-spread, provides a comprehensive set of diagrams for both structural and behavioral modeling, and supports the early phases of process modeling (analysis and design).
Like many other object-oriented modeling languages, UML has no well-defined semantics. We indicate how a process model described in UML can be automatically transformed into an executable form, i.e., we provide dynamic semantics for UML models. To this end, UML models are transformed into programmed graph rewriting systems which are used to drive a process management environment.

Keywords: Software Process Models, Software Engineering Tools and Environments

1 Introduction

Software processes are highly dynamic. Many changes have to be taken into account while a software process is being executed: changing requirements, feedback to earlier stages of the software life cycle, moved deadlines, shrinking budget, etc. These changes challenge the capabilities of process-centered environments [7].

For modeling software processes, we have proposed *dynamic task nets* [9], i.e., hierarchies of tasks that are in addition connected by various kinds of horizontal relationships (control flow, data flow, and feedback). The most essential feature of these task nets is that they continuously evolve during the enactment of a software process. This contrasts sharply to the distinction between build time and run time, as it is made in most workflow management systems [16].

Originally, dynamic task nets were defined without any reference to an object-oriented modeling approach. However, the continuous evolution of task nets makes an object-oriented approach particularly attractive. Therefore, we have decided to examine the benefits of using an object-oriented modeling language for software process modeling. For this purpose, the *Unified Modeling Language* [3] appeared to be a natural choice. Some of the benefits we expected include:

O. Nierstrasz, M. Lemoine (Eds.): ESEC/FSE'99, LNCS 1687, pp. 91–108, 1999.

- If a wide-spread notation is used, process models can be communicated more easily to a larger number of people.
- UML provides a large set of diagrams (class diagrams, object diagrams, collaboration diagrams, state diagrams, etc.) which can be used to define both structure and behavior of dynamic software processes.
- Object-oriented modeling supports the earlier phases of process engineering (analysis and design), while most process modeling approaches, in particular those underlying process-centered environments, primarily focus on process programming.

On the other hand, we also expected some problems, in particular because UML is an informal modeling language which does not have a well-defined (dynamic) semantics.

In this paper, we describe how we are using UML for modeling software processes based on dynamic task nets. Section 2 summarizes the main features of dynamic task nets, Section 3 introduces the main components of the DYNAMITE process management environment. Section 4 constitutes the main part of this paper, which is devoted to process modeling in UML. In Section 5, we define the semantics of UML process models by a mapping into a graph rewriting system. Section 6 summarizes the lessons learned. Related work is compared in Section 7. Finally, Section 8 presents a short conclusion.

2 Dynamic Task Nets

The *DYNAMITE* model [9] introduces DYNAMIc Task nEts for software process management. A *task* represents a unit of work that is typically performed by a human developer (with tool support). A task may have *input parameters* and *output parameters*, which are the documents the task is working on. A task reads its input documents upon activation, works with these documents and finally produces some output documents. Tasks are connected by *control flow* relationships which describe the order of execution of tasks. In addition, there are *data flow* relationships which refine control flow relationships. While control flow state only the existence of temporal dependencies, data flows describe the passing of documents between tasks.

Figure 1 shows a dynamic task net which models the process of extending a software system during maintenance. At the beginning only little is known about the process. The request for extending the software system has to be analyzed and the application has to be redesigned. Finally, the modified system has to be installed. The intermediate structure remains unspecified, because it depends on the changes to the design document's internal structure (part i). According to the new design document produced by the task Redesign Application, modules B and D have to be changed and a new module C has to be introduced. Thus, the project manager introduces two tasks for changing modules B and D together with a task for implementing module C. In addition, he adds subsequent test tasks. The control flow relationships between the test tasks reflect the module hierarchy, i.e., the topmost module D is the last one to be tested (part ii).

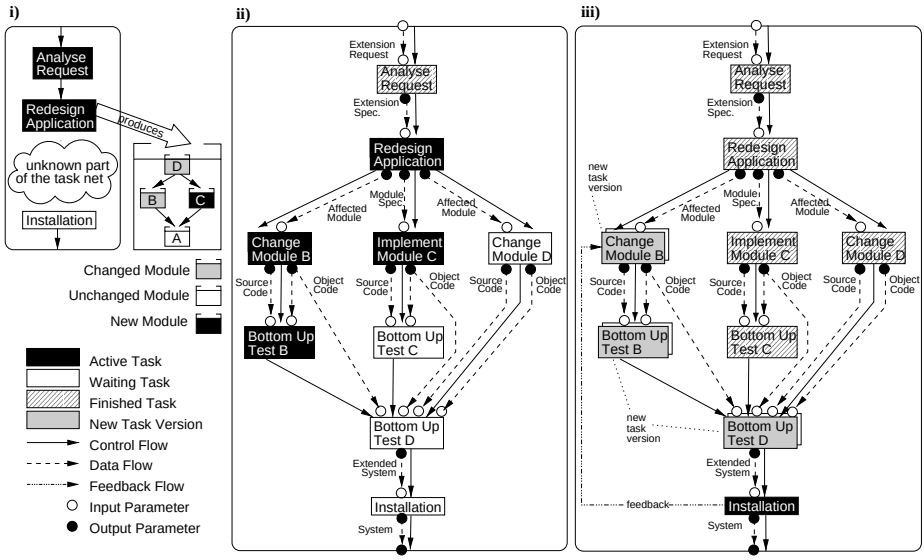


Fig. 1. Examples of dynamic task nets

The execution of tasks is modeled by their *states*. E.g., a task which is currently being executed is in state Active. Note that a task may start execution before termination of its predecessors (simultaneous engineering). Tasks which wait for their execution are in state Waiting. After a task has produced all of its output parameters, its execution is finished and its new state is Done.

Faulty output documents produced by predecessor tasks may cause errors in later phases of the development process. In this case, it is necessary to go back to the predecessor task and produce a corrected version of the document. This situation is modeled by a *feedback flow* which is introduced between the task where the error was detected and the task which is responsible for it. If the predecessor task has already terminated, it is not reactivated. Rather a new *task version* is derived from the old one and the old work context is reestablished. Using derivation rather than reactivation of tasks results in creation of a process execution's trace, which can later be examined by the project manager to find indications for process improvement. For example, in part iii of Figure 1 Installation raises feedback to Change Module B, resulting in new task versions along the path from the feedback's source to the feedback's target.

3 Environment Overview

The DYNAMITE environment supports human-centered development processes through various tools (Figure 2). Humans interact with these tools in three different roles: as project manager, technical developer or as process modeler.

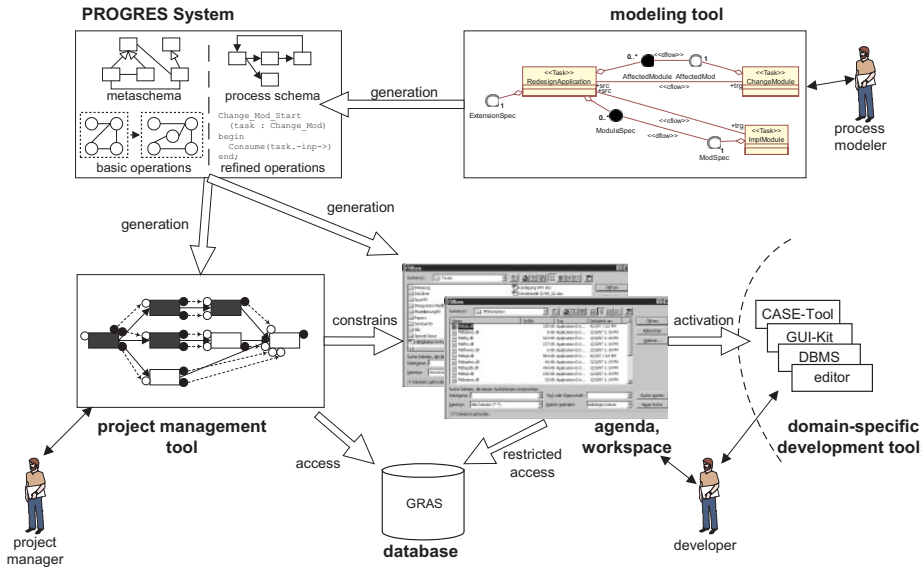


Fig. 2. Overview of the DYNAMITE environment

The *project management tool* provides unrestricted access to a task net. It supports a *project manager* in constructing and analyzing task nets. During enactment he is provided with mechanisms and views to guide and monitor process performance. Internally, task nets are stored in a *graph-based database system*.

The *technical developers* enacting the development process are supported by *agenda and workspace tools* that allow restricted, personalized access to a task net. An agenda tool deals with the tasks that have been assigned to a developer. It provides information on deadlines, status of tasks, guidelines etc. and is a developer's central access structure to the management environment. When working on a specific task, a workspace is automatically provided. It gives access to versioned documents relevant for performing the task. Domain-specific *technical development tools* are integrated into the environment and can be activated on selected documents provided by the workspace tool.

For the internal operational definition of dynamic task nets we utilize the graph transformation system *PROGRES* (PROgrammed Graph REwriting Systems [24]). Graph transformations are a suitable approach because task nets form complex graph data structures and operations to manipulate and enact task nets can be specified using a uniform mechanism. This enables intertwined editing and enacting of task nets as was presented in Section 2.

In a PROGRES specification the metaschema of dynamic task nets is defined as a graph type consisting of node types, edge types, and attributes. A set of generic operations to manipulate and enact task nets is provided as complex graph transformations and procedures of these. This generic part of the specifi-

cation deals with untyped task nets, where all tasks show a predefined standard behavior. This base specification can be used for *ad hoc workflows* in the case of chaotic processes. Moreover, executing ad hoc workflows may provide us with the process knowledge to be gathered for process analysis so that we may ultimately develop a more structured process model.

In order to feed domain-specific process knowledge into the environment the specification can be enhanced in two ways. On the one hand, a specific process schema as an extension to the existing metaschema may be defined to provide structural constraints for the evolving task nets. On the other hand, the generic operations may be refined in order to introduce specific process execution policies like simultaneous or sequential engineering.

While PROGRES is a very suitable environment to declaratively specify the generic model, it can hardly be offered a *process modeler* as a *process modeling tool*. A process modeler is usually no expert on graph transformations. For this reason we decided to enrich our DYNAMITE environment with a modeling tool which offers a process modeler comprehensive languages for process modeling. To this end we use the Unified Modeling Language (UML) (cf. Section 4). The domain-specific part of the PROGRES specification is then generated from the UML model (cf. Section 5).

Using PROGRES' C-code generation facilities, the management tool's functional core is generated from the resulting PROGRES specification. To build a proper tool with a graphical user interface we make use of a framework for graph based applications that ships with the PROGRES environment [12].

Introducing a modeling tool serves multiple purposes: The modeling task is simplified significantly. Process models can be expressed quite naturally and can thus be better communicated to others and reasoned about. Building a process modeling tool allows us to offer online semantic analyses to guide the process modeler and enables us to support reuse of model fragments.

4 Process Modeling in UML

To increase model understandability and to decrease model maintenance effort, process modeling should be supported on a very high level of abstraction. In contrast to other approaches introducing a special purpose process modeling language (EPOS [10], JIL [25], MADAM [21]) this section will demonstrate how process models can be adequately defined in a standardized object-oriented modeling language, the UML.

4.1 Structural Modeling

The process specification models processes on the type level. Resulting *process schemas* thus abstract from a multitude of instance level task nets (examples of which were presented in Section 2). A process schema's structure consists of task, parameter and realization classes and their various interdependencies. Consequently, we use UML's *class diagrams* to model this structure.

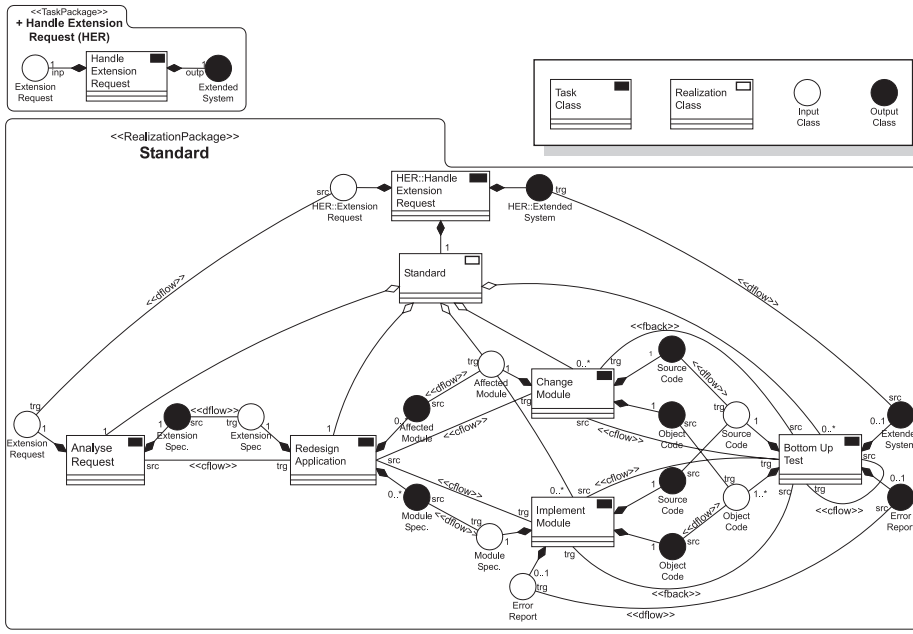


Fig. 3. Task interface and realization

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 a set of possible realizations, one of which can be selected by the corresponding actor at process enactment time.

Figure 3 shows the interface of the task class Handle Extension Request at the top. We make explicit use of UML's *stereotype* concept and introduce stereotypes for task classes (symbolized by a black rectangle inside of the class box), and input and output parameter classes (symbolized by a white and black circle, respectively). Stereotypes are used to express the underlying process meta model (DYNAMITE).

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. Our usage of UML's package concept is explained in detail later on.

A task class composes all its respective realization classes (symbolized by a white rectangle). Since the interface abstracts from a set of possible realizations, these realization classes are not part of the interface of a task package. Rather an individual package is introduced for every realization class. Realization classes abstract from complex schematic realizing subprocess definitions as shown in the bottom part of Figure 3. These allow for the *composition* of other task classes through the following associations: *Control flow* associations introduce a tem-

poral ordering on tasks and are similar to ordering relations in PERT-charts. *Feedback flow* associations are always directed oppositely to control flow associations and are introduced to mark iteration or exception steps of a process. The direction of these associations is indicated through the role names src and trg. To make these associations distinguishable, we introduce corresponding stereotypes, namely ≪cflow≫ and ≪fback≫.

These association types are sufficient to model the flow of control in a group of tasks. For example, the Analyse Request and Redesign Application classes are connected by a control flow association which indicates that application redesign cannot take place before analysis of the request (however, execution may overlap). Analogously, feedback is allowed between the Bottom-Up Test and the Implement Module or Change Module classes in case of errors during the test. Of course, feedback could equally well occur in other parts of this subprocess model, which is not shown here.

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. Equally, they may be introduced between two input or two output classes if the corresponding task classes are hierarchically related. This gives the complex parent task the possibility to supply the refining tasks with its input data and to receive the results from the refining subprocess.

In our example the Analyse Request task is supplied with the initial extension request. There, the request is transformed into an extension specification, which is subsequently sent to the Redesign Application task. After all modules have been changed or implemented and successfully tested, the Extended 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 various classes at process enactment time. In the example the number of Analyse Request and Redesign Application tasks is restricted to exactly one, while Change Module and Implement Module tasks may occur in any number at enactment time. Their actual number will depend on the number of modules that have to be changed or implemented from scratch, respectively.

4.2 Model Structuring

The structuring of resulting models is vital for model comprehension and reuse. We use the UML's package concept for this purpose. Packages are utilized to group closely related modeling elements together, to offer a separate name space, and to define visibilities. Each task and realization definition is stored in a separate package. These packages are distinguished by their respective stereotypes ≪TaskPackage≫ and ≪RealizationPackage≫. Figure 3 shows a sample *task* and *realization package*.

It remains to be explained how packages can be interrelated and what kind of visibility has to be defined for them. To establish a flexible model structuring

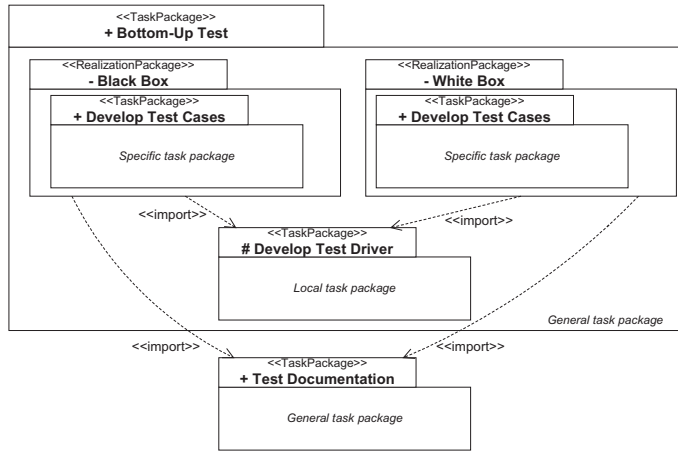


Fig. 4. Usage of packages

concept, the scope of task classes for a model has to be pointed out. For this purpose we distinguish three categories of task classes:

1. A task class' scope can exceed one process model. These task classes are highly reusable and will be called *general* task classes in the following. An example for a general task class is one for the software test which may be used in process models for software extension, correction, or initial development.
2. Other task classes are very specific and only relevant for one realization class alone. They will be denoted as *specific* task classes. If we look at a scenario where bottom-up testing can alternatively be performed as a black- or a white-box test, each of the corresponding realization packages will naturally contain a different task class for the development of test cases.
3. A third set of task classes is characterized by their relevance for various realization classes of the same task class. Their scope is thus local to one task class and they will be called *local* task classes in the following. If an alternative to bottom-up testing is top-down testing, a task class for test driver development will only be needed for bottom-up testing. However, it will be needed regardless of how the bottom-up test will be refined. Thus, it is declared as a local task class for bottom-up testing.

If we use the same categories for the task packages containing task classes, we can derive a suitable model structuring concept from this categorization (cf. Figure 4). Local task packages may be nested into general or other local task packages. Every task package contains its respective realization packages while each realization package contains its specific task packages.

To reach abstraction and information hiding within process models, adequate *visibilities* have to be defined [22]. The visibility of realization packages is inherently private (-), because only the interface of a task class needs to be known to potential users. General task packages are provided with a visibility of public (+)

since their contained task interface can be used in any context. The offered task interface of local task packages should in contrast only be visible to realization packages nested into the same task package but not to importers of the latter. For this reason, the visibility of local task packages has to be protected (#). Specific task packages are nested into realization packages. Since their visibility is private no package outside of a realization package can access the contained task packages. The visibility of specific task packages will thus be set to public (+), which allows the superior realization package to use the offered task interface.

The advantages of using UML packages in the afore mentioned way are manifold. They allow for the modular modeling of processes so that complex processes are separated into manageable fragments and loose coupling of models is reached. In addition, reusable process fragments can be identified since general and local task packages are subject to reuse. The concept introduces abstraction and information hiding in several ways, which allows for the manipulation and exchange of process model fragments with little impact on the overall model.

4.3 Behavioral Modeling

Every task class owns a *generic set of methods* and an attribute for the current execution state. The interrelations of these methods are modeled in a prescribed *state diagram* (Figure 5, part i). Methods can be subdivided into the set of state changing methods (e.g., Start, Plan, Suspend), the set of data flow related methods (e.g., Consume, Produce, Release) and the set of methods to edit task nets (e.g., CreateTask, CreateParameter, CreateControlflow). The figure shows all state-changing methods but gives only examples of state-preserving ones.

Prescribing one state diagram for all task classes may seem a little restrictive, but allowing the modeler to define a specific state diagram for every task class would imply the need to model the interrelations between state diagrams for every pair of task classes. The resulting increased complexity would counteract our aim to simplify the task of modeling a process. Experience has shown that the state diagram of Figure 5 is sufficient to adequately model a software process.

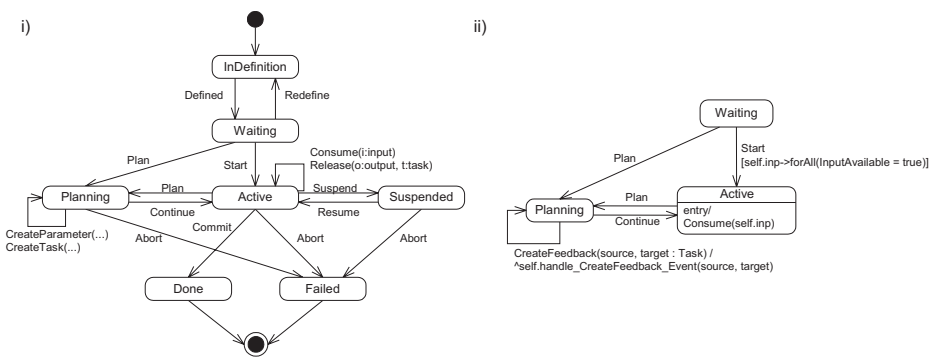


Fig. 5. (i) Generic and (ii) (cut-out of) specific state diagram

The state diagram allows tasks to be currently in definition, to be waiting for activation, to be active, suspended, or (re-)planned. Every task can terminate in one of two final states, one of which marks its successful completion (Done), while the other one marks its failure (Failed).

Within the underlying process engine some *standard behavior* is defined with respect to this state diagram. Methods consider the states of preceding and succeeding tasks as well as of subordinate and superior tasks. For example, a task may only terminate when all predecessors have terminated. In addition, the suspension of a task leads to the suspension of all subordinate tasks.

Every method being executed by a task by default sends out a corresponding event to its predecessors, successors, children, parent, and itself. 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 respective context.

This standard behavior can be adapted by the process modeler in three different ways. The formulation of *transition conditions* in the *Object Constraint Language* (OCL) provides the means to influence the way a task net is executed. Furthermore, UML allows for the definition of action calls inside of a state to automate execution steps. In addition, an action can be triggered by entering or by leaving a state. Finally, the set of event-receiving tasks can be restricted by using UML's send-clauses as part of transition definitions.

A cut-out of a sample adapted state diagram of the Handle Extension Request task class is shown in part ii of Figure 5. The start transition can only be fired when all inputs for the task are available (the standard behavior does not demand this). In addition, all inputs can now be automatically consumed when a task of this type enters the state Active, since it is ensured that they are available. Furthermore, the sample state diagram restricts the target set of the CreateFeedback event to the task itself. In this case it makes sense to restrict the standard behavior of sending an event to the whole context, since we want this event to be handled by a complex task net transformation (see below).

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. UML allows to specify a method's semantics in any language, like C++ oder pseudo code. For example, the automatic activation of a task, dependent on 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 owning task. This way events can lead to very complex *task net transformations* since the structure of the subprocess is known to the realization class.

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 avail-

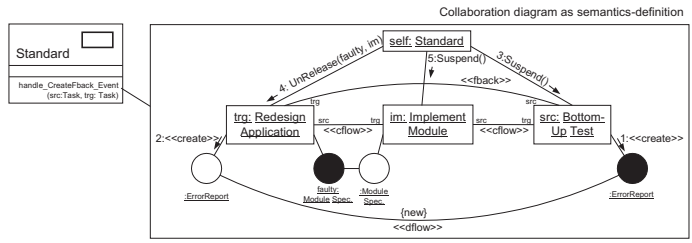


Fig. 6. Complex event handler

able 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 6 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 Redesign Application. 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: an output parameter at the Bottom Up Test task and an input parameter at the Redesign Application task. Creation of new objects is done through sending a ≪create≫ message (see messages numbered 1 and 2 in Figure 6). 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}. Finally, the tasks' behavior is specified through messages. In the example case, the feedback flow's source is suspended (message 3), the erroneous output document of the feedback flow's target is retracted (message 4), and the intermediate implementation task is suspended (message 5).

Specifying the behavior through state diagram adaptations and the definition of custom event handlers may still be too work intensive. Experience has shown that a limited set of behavioral patterns is used for most behavioral specifications. Examples for behavioral patterns are sequential and concurrent engineering as properties of control flow associations. To simplify the modeling of specific behavior, we propose the use of one of UML's extension mechanisms, the *tagged values*. For example a tagged value of name policy could be defined for the stereotype ≪cflow≫ with possible values of sequential and concurrent. Our eventual goal is to come up with a library of predefined behavioral patterns which may be combined easily at a high level of abstraction.

5 Generating Executable Process Models

UML itself is not *executable*, i.e., process models described in UML can neither be simulated nor enacted. However, they can be transformed into an executable form as follows.

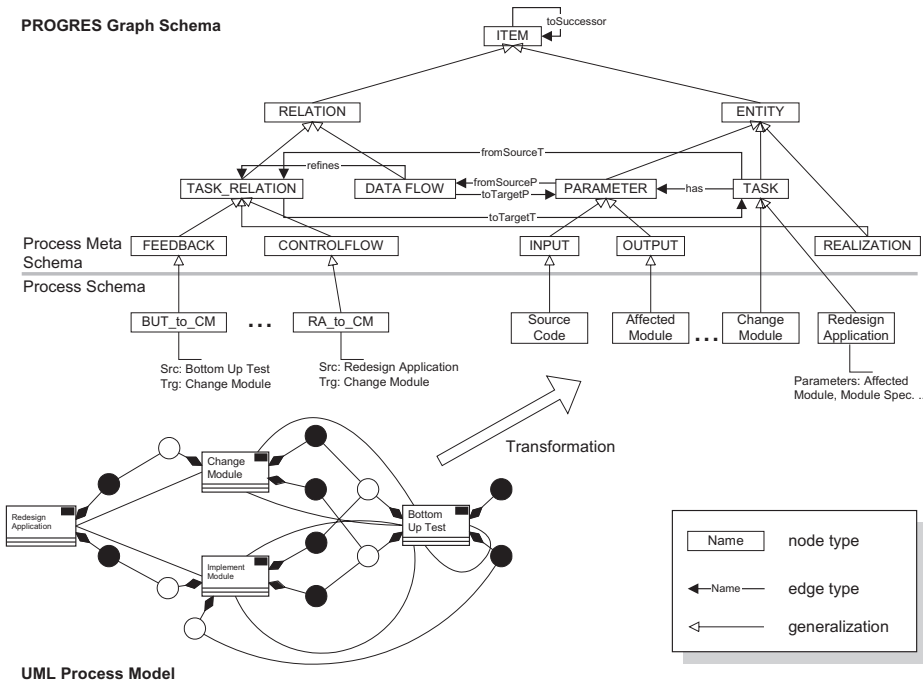


Fig. 7. Graph schema extension

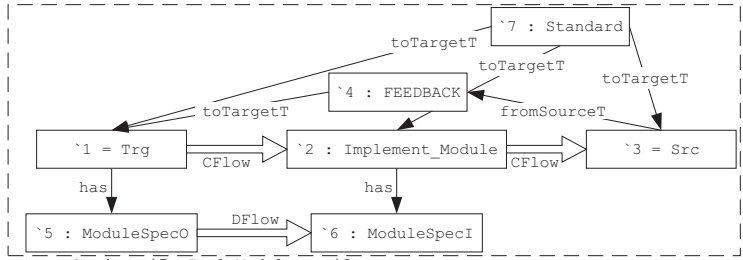
A process specification in UML is transformed into a corresponding specification in PROGRES which extends the base specification (cf. Section 3). UML class diagrams are transformed into a graph schema extension defining domain-specific task and relation types (Figure 7). The figure shows a cut-out of the *generic graph schema* which defines a process meta schema. It contains node types as abstractions of process objects (TASK, INPUT,...) and process object relations (FEEDBACK, DATAFLOW,...). The latter have to be modeled as node types because they carry attributes which are important for the enactment and manipulation of instance level task nets and PROGRES does not support attributed edges.

The *domain specific process schema* which is modeled in UML is transformed into node types as an extension to the generic graph schema. The modeled UML associations and aggregations are mapped to node types carrying schema-level attributes as shown in Figure 7 for the source and target types of task relations. This way the instantiation of the meta schema's edge types is restricted according to the UML model.

State transitions, automated action calls, and event dispatchers are transformed into procedures calling the generic model's base operations.

Graphical definitions of event handlers as presented in Figure 6 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

```
test Thandle_CreateFeedback_Event( Src : Bottom_Up_Test;
                                   Trg : Redesign_Application;
                                   out Strd : Standard;
                                   out Impl_Module: Implement_Module;
                                   out FB : FEEDBACK;
                                   out Mspec1 : ModuleSpecO;
                                   out Mspec2 : ModuleSpecI; )
=
```




```
  return Strd := `7; Impl_Module := `2;
         FB := `4 ; Mspec1 :=`5; Mspec2 := `6;
end;

transaction handle_CreateFeedback_Event(Src, Trg : TASK) =
  (* declaration of local variables for process objects*)
    Thandle_CreateFeedback_Event(Src : Bottom_Up_Test,
                                 Trg : Redesign_Application, ...)
  & CreateParameter(Src, Error_ReportO, out ER1)
  & CreateParameter(Src, Error_ReportI, out ER2)
  & CreateDataflow(ER1, ER2, FB)
  & Suspend(Src)
  & UnRelease(Mspec1, Impl_Module)
  & Suspend(Trg)
end;
```

Fig. 8. Transformed event handler

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 base operations of the generic model. All message calls to objects (i.e. messages 3-5 in Figure 6) are transformed into corresponding calls to base operations.

Search and replacement of an 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. This circumstance is also known as the *graph rewriting dilemma* (violation of data abstraction).

Figure 8 shows a sample graph query and procedure (called *transaction* in PROGRES) for the collaboration diagram in Figure 6. 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.

Since currently only one unstructured graph schema is supported by PROGRES, the provided package structure of the UML model is lost during transformation. However, a module concept for PROGRES is currently being developed [23] which will enable to preserve the structure in the transformed model.

Technically, the transformation from UML models to PROGRES is performed in the following way: We use the commercial CASE tool Rational Rose as a modeling tool. Rose offers mechanisms to access its operations and the current model through a Component Object Model (COM) interface. The implementation of the transformation makes use of this interface to read the current model and generates the PROGRES-Code as a text file.

Note that no manual implementation is involved here. Rather, the UML specification is transformed automatically into a PROGRES specification (which is translated into C code by the PROGRES compiler). In this way, we combine a high-level, executable specification language with an object-oriented modeling language constituting an emerging standard that will be used widely.

6 Lessons Learned

Object-oriented software process modeling. Modeling tasks as objects is quite natural. Dynamic task nets are created, modified, analyzed, and executed during the course of a software project. Therefore, a task net can be represented as an evolving object structure on which different components of a process management environment operate (e.g., editor, planner, analyzer, or execution tool).

UML provides a rich set of diagrams for modeling both the structure and the behavior of dynamic software processes. In this paper, we have focused on the use of class diagrams for structural modeling and the use of state diagrams and collaboration diagrams for behavioral modeling. In addition, we have applied the UML package concept to structure process models (modeling-in-the-large). Due to the lack of space, we have described only a subset of the diagrams which we actually use in our approach. For example, we employ both use case diagrams and object diagrams for process analysis [13].

However, we do not exploit the full range of diagrams offered by UML. In particular, we have dismissed activity diagrams even though they were added to UML for business process modeling. Unlike dynamic task nets, activity diagrams are statically defined at modeling time. Since activities are not modeled as objects, it is not possible to model the dynamics of software processes at enactment time.

Adapting UML. It is important to understand that we do not simply apply UML as it is. Rather, we adapt UML according to our process meta model. To this end, we employ the extension mechanisms offered by UML: stereotypes, constraints, and tagged values.

Unfortunately, these mechanisms support metamodeling only to a limited extent. Rather, we would like to use UML itself as a full-fledged metamodeling language. The syntactical structure of dynamic task nets could then be defined in a class diagram and constraints could be added to define their static semantics. However, UML does not provide a layer for building domain-specific meta models in this way [18]. Note that defining a meta model by extending UML's own meta model is an unsatisfactory approach. While adding new base classes increases UML's modeling capabilities for a domain [27], it does not restrict UML with

regard to the domain's modeling constraints. In addition, concerns are not well separated.

From informal to formal process models. As such, UML is an informal language in that process models written in UML cannot be enacted. In contrast, process modeling languages for process-centered software engineering environments must be enactable, i.e., a process model must have well-defined execution semantics.

As described in Section 5, we have solved this problem by defining a mapping from UML process models to programmed graph rewriting systems. The mapping is partial because UML models must conform to the underlying process meta model. It is also deterministic, i.e., no user interaction is required to generate PROGRES code from a UML model.

It should be noted that we have not attacked the problem of defining generally accepted semantics for the UML. We have restricted ourselves to those parts of UML which are relevant for our process modeling approach. In addition, we define the semantics according to our process meta model.

UML as a unified process modeling language? In response to the large number of object-oriented modeling approaches, UML has emerged as a standard notation which makes communication easier. Instead of learning many different notations for the same concept, modelers may stick to just one wide-spread standard notation.

In software process modeling, researchers tend to define their own languages even if their underlying concepts could be expressed in a standard notation. So we decided to investigate the use of a standard notation for process modeling. So far, we consider our experiment as successful. Class diagrams, state diagrams, collaboration diagrams etc. provide us with well-suited modeling elements for software process modeling. By using a wide-spread modeling language, we hope to increase the acceptance of our process modeling approach — and to leverage the communication with its users, who may not have any background in software engineering at all (note that we apply dynamic task nets also in mechanical and chemical engineering [17]).

However, concerning the unification of process modeling approaches, we are not too optimistic. First, many process modeling languages are not object-oriented or cannot be naturally expressed in an object-oriented modeling language such as UML (e.g., rule-based languages [2]). Second, UML can be applied in radically different ways to process modeling. Even when the same notation is used, the underlying process meta models may vary considerably.

7 Related Work

Throughout this paper, we have been concerned with process modeling in UML. We have not discussed process modeling for UML, i.e., methods for applying UML to software development [11]. Our approach provides a fairly general meta model which can be applied to a wide range of processes — not only in software engineering, but also in other domains.

Process modeling in UML is often performed with the help of activity diagrams [28,14]. Activity diagrams have been introduced for modeling business processes, which causes several problems when being applied to software processes. Activity diagrams are viewed as process programs, even though software processes are hard to predict. Modeling processes as objects is more adequate since it allows for representing process evolution.

The software process community has developed a wide range of process modeling languages. By and large, researchers preferred developing their own languages rather than using wide-spread object-oriented notations. An exception to this is ESCAPE+ [19], which is partially based on OMT and employs class and state diagrams for process modeling. However, ESCAPE+ focuses on process specification and does not address process analysis. Moreover, the underlying process meta model differs considerably from DYNAMITE (e.g., ESCAPE+ models tasks as operations attached to document objects, while DYNAMITE models them as first-class objects). Further object-oriented languages for software process modeling not specifically committed to standard notations are E3 [1] and SOCCA [5].

Let us now compare the process meta model to other work at a conceptual level, ignoring the notations used (note, however, that usage of a standard notation is the essential contribution of this paper). In many approaches, process models are viewed as programs to be executed at run time. This *direct execution* paradigm is realized e.g. in modeling languages based on procedural programming [26] or Petri nets [4]. In contrast, we follow an *indirect execution* paradigm: the (program) task net to be executed is constructed only at run time. In this respect, our approach is similar to rule-based approaches, where a plan is constructed dynamically [10]. However, we believe that task nets can be created at best semi-automatically; in particular, project managers need the ability to build up and modify task nets manually.

Our approach combines UML with graph rewriting. In [8], a specification language (Fujaba) is described which combines UML and graph rewriting as well. However, our intent is not to introduce a new specification language. Rather, we focus on the application of an existing modeling language (UML) to process modeling and define the semantics of UML models by an application-specific mapping from UML to PROGRES.

8 Conclusion

We have demonstrated the use of UML as a process modeling language. Moreover, we have shown how UML can be employed as a "front end" to a process management environment. To this end, a formal interpretation for UML process models is provided through a translation into PROGRES.

Using UML for software process modeling provides us with several benefits. First, dynamic task nets lend themselves quite naturally to object-oriented modeling. Second, UML provides a rich set of diagrams for describing process models. Third, it assists the earlier phases of process model development. Fourth, we

may attach semantics to UML process models by mapping them to programmed graph rewriting systems. However, we have also discovered some problems, particularly concerning the selection of a useful subset of diagrams and the adaptation of UML (missing metamodeling facilities). Finally, we acknowledge the advantages of using a standard notation, but we also have some reservations with respect to the degree of unification in software process modeling that can be achieved in this way.

Before having switched to UML as a process modeling language, we have designed MADAM (Management and ADaptation of Administration Models), a high-level language consisting of both graphical and textual elements [21]. We have also implemented a modeling environment supporting the creation of MADAM models and their translation into a programmed graph rewriting system (the details of this translation can be found in [15]). The translation of UML process models has been performed according to the same philosophy.

The process management environment (DYNAMITE) is currently being redesigned and -implemented [12]. In particular, we are going to enhance the user interface with the help of a commercial tool-kit (ILOG JViews), and we have selected Java as our new implementation language (formerly, we used C and Tcl/Tk).

References

1. M. Baldi, S. Gai, M. L. Jaccheri, and P. Lago. E3: Object-oriented software process model design. In Finkelstein et al. [7], pages 279–292.
2. N. S. Barghouti and G. E. Kaiser. Scaling up rule-based software development environments. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):59–78, Mar. 1992.
3. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, Reading, Massachusetts, 1998.
4. 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.
5. G. Engels and L. Groenewegen. SOCCA: Specifications of coordinated and cooperative activities. In Finkelstein et al. [7], pages 71–102.
6. G. Engels and G. Rozenberg, editors. *TAGT '98 — 6th International Workshop on Theory and Application of Graph Transformation*, number tr-ri-98-201 in Series in Computer Science, Paderborn, Germany, Nov. 1998. Department of Computer Science, University of Paderborn.
7. A. Finkelstein, J. Kramer, and B. Nuseibeh, editors. *Software Process Modelling and Technology*. Research Studies Press (John Wiley & Sons), Chichester, UK, 1994.
8. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph grammar language based on the unified modeling language and Java. In Engels and Rozenberg [6], pages 112–121.
9. 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.

10. M. L. Jaccheri and R. Conradi. Techniques for process model evolution in EPOS. *IEEE Transactions on Software Engineering*, 19(12):1145–1156, Dec. 1993.
11. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Object Technology Series. Addison-Wesley, Reading, Massachusetts, 1999.
12. D. Jäger. Generating tools from graph-based specifications. In J. Gray, editor, *Proceedings First International Symposium on Constructing Software Engineering Tools*, pages 97–107, Los Angeles, May 1999. University of South Australia, School of Computer Science.
13. D. Jäger, A. Schleicher, and B. Westfechtel. Modeling dynamic software processes in UML. Technical Report 98-11, RWTH Aachen, 1998.
14. A. Korthaus. Using UML for business object based systems modeling. In Schader and Korthaus [20], pages 220–237.
15. C.-A. Krapp. *An Adaptable Environment for the Management of Development Processes*. Number 22 in Aachener Beiträge zur Informatik. Augustinus Buchhandlung, Aachen, Germany, 1998.
16. P. Lawrence, editor. *Workflow Handbook*. John Wiley & Sons, Chichester, UK, 1997.
17. M. Nagl and B. Westfechtel, editors. *Integration von Entwicklungssystemen in Ingenieuranwendungen*. Springer-Verlag, Heidelberg, 1998.
18. Rational. *UML Semantics*, 1.1 edition, Sept. 1997. http://www.rational.com/uml.
19. W. Reimer, W. Schäfer, and T. Schmal. Towards a dedicated object oriented software process modelling language. In *Object-Oriented Technology — ECOOP '97 Workshop Reader*, LNCS 1357, pages 299–302, Jyväskylä, Finland, June 1997.
20. M. Schader and A. Korthaus. *The Unified Modeling Language — Technical Aspects and Applications*. Physica-Verlag, Heidelberg, Germany, 1998.
21. 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.
22. A. Schürr and A. Winter. Formal definition of UML's package concept. In Schader and Korthaus [20], pages 144–160.
23. A. Schürr and A. Winter. UML packages for programmed graph rewriting systems. In Engels and Rozenberg [6], pages 132–139.
24. 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.
25. S. Sutton and L. Osterweil. The design of a next generation process language. In M. Jazayeri and H. Schauer, editors, *Proc. ESEC '97*, LNCS 1301, pages 142–158, Zürich, Switzerland, Sept. 1997.
26. 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.
27. P. S. Tom Mens, Carine Lucas. Supporting reuse and evolution of UML models. In J. B. Pierre-Alain Muller, editor, *Proceedings of UML'98 International Workshop*, pages 341–350, Mulhouse, France, 1998.
28. G. Versteegen. Objektorientierte Geschäftsprozeßmodellierung mit der UML: die Innovator Business Workbench. *OBJEKTspektrum*, (1):62–67, Jan. 1998.