Modeling Dynamic Software Processes in UML

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

Abstract

We propose the usage of the standard notation UML for process modeling rather than a dedicated process modeling language.

We regard processes as products of project management. Consequently, our approach is object-oriented and does not model processes through activity diagrams. Activity diagrams have been introduced for modeling business processes which causes several problems. Activity diagrams are viewed as process programs, even though software processes are hard to predict. Resulting modifications tend to increase the diagrams' complexity which reduces common comprehension due to numerous alternatives.

In contrast, our approach starts out with use cases and informal descriptions of processes. We gather knowledge on sample processes in collaboration diagrams showing task net snapshots. We generalize the structure of processes in class diagrams. This leads us to a schematic view of the process. The structural constraints we impose on class diagrams enforce the construction of loosely coupled process patterns. These patterns are stored inside UML packages and are subject to reuse.

The behavior of tasks within the process is specified by state diagrams and collaboration diagrams. This allows for the specification of multiple execution patterns for e.g. sequential or concurrent engineering, the automated execution of task nets and complex task net transformations.
Keywords: process modeling, UML, process management

1 Introduction

Process modeling has become a very active area of research in many different domains, including e.g. software engineering, mechanical or electrical engineering, or office automation. Many approaches to process modeling have been proposed. For example, there is a great variety of process modeling languages for software engineering [4, 5], based on Petri nets, rules with pre- and postconditions, graphs, attribute grammars, state machines, procedural process programs, events and triggers, etc. In fact, this has resulted in a Babylonian-like confusion. Understanding, communication, classification, comparison, and assessment of all of these approaches appears to be virtually impossible.

In object-oriented modeling, the UML — *Unified Modeling Language* [15, 16] — has been introduced recently as a uniform notation. Applying UML to process modeling promises several benefits. First, a uniform notation makes different approaches more easily comparable. Second, process models described in UML can be communicated to a larger number of people. Third, usage of UML puts more emphasis on the earlier phases of the meta process, while many current process modeling languages primarily focus on process programming.

Although we recommend the use of a standard notation, we are aware that there are still different ways to apply UML to process modeling. One way is to use *activity diagrams*, which are composed of nodes representing activities and edges representing control flows. Activity diagrams support procedural modeling of processes, according to the process programming paradigm of Osterweil [14], which is also implemented in many workflow management systems [13]. In addition to activities, activity diagrams may contain nodes for object states (pre- and postconditions of activities). Moreover, we may express organizational responsibilities for activities. Altogether, this results in a comprehensive modeling approach including activities, objects, and actors. In the sequel, we will call this approach *activity-centered*.

The major drawback of the activity-centered approach is that activities are not represented as objects. In contrast, we regard processes as products of project management. More specifically, at least in software engineering we can rarely predict a software process so accurately that we may encode it as an activity diagram to be executed subsequently under the control of some process engine. In contrast, the process evolves during execution. For example, the tasks to be executed may depend on the product structure, which is fixed only during development; feedback may occur requiring the re-execution of activities, etc. The dynamics of software processes, which is particularly prominent in concurrent engineering [17], demands for an *object-centered* modeling approach, i.e., activities are represented as objects (instances of classes). Operations on these objects include creation, start, suspension, termination, consumption of input data, production of output data, etc.

We apply UML to process modeling at different levels. First, *process analysis* captures process knowledge in an informal way. Use cases describe processes in terms of diagrams showing the processes and the actors involved in their execution. Each use case is further described by a set of object diagrams representing sample processes for that use case. Second, *process specification* generalizes the examples gathered during process analysis and describes them in a formal way. To this end, we employ class diagrams and statechart diagrams for structural and behavioral modeling, respectively.

Process modeling is performed in a conceptual framework — a *process meta model* — which is encoded with the help of extension mechanisms offered by UML. By means of these mechanisms (stereotypes and properties), we introduce modeling elements such as tasks, inputs and outputs, data and control flows, etc. The process meta model is based on DYNAMITE, which is an acronym for <u>DYNAMI</u>c <u>T</u>ask N<u>E</u>ts [6].

In addition, DYNAMITE stands for a *process management environment* [7] which is used to support the execution of software processes. To this end, process models encoded in UML are transformed into executable process models for the DYNAMITE environment. In this way, process modelers are shielded from the formalism for defining executable process models (the specification language PROGRES [20], which is based on programmed graph rewriting systems).

2 Meta Process

For the description of the *meta process* of process modeling we use an informal notation that is easy to understand and requires no knowledge of the UML (just to get started).

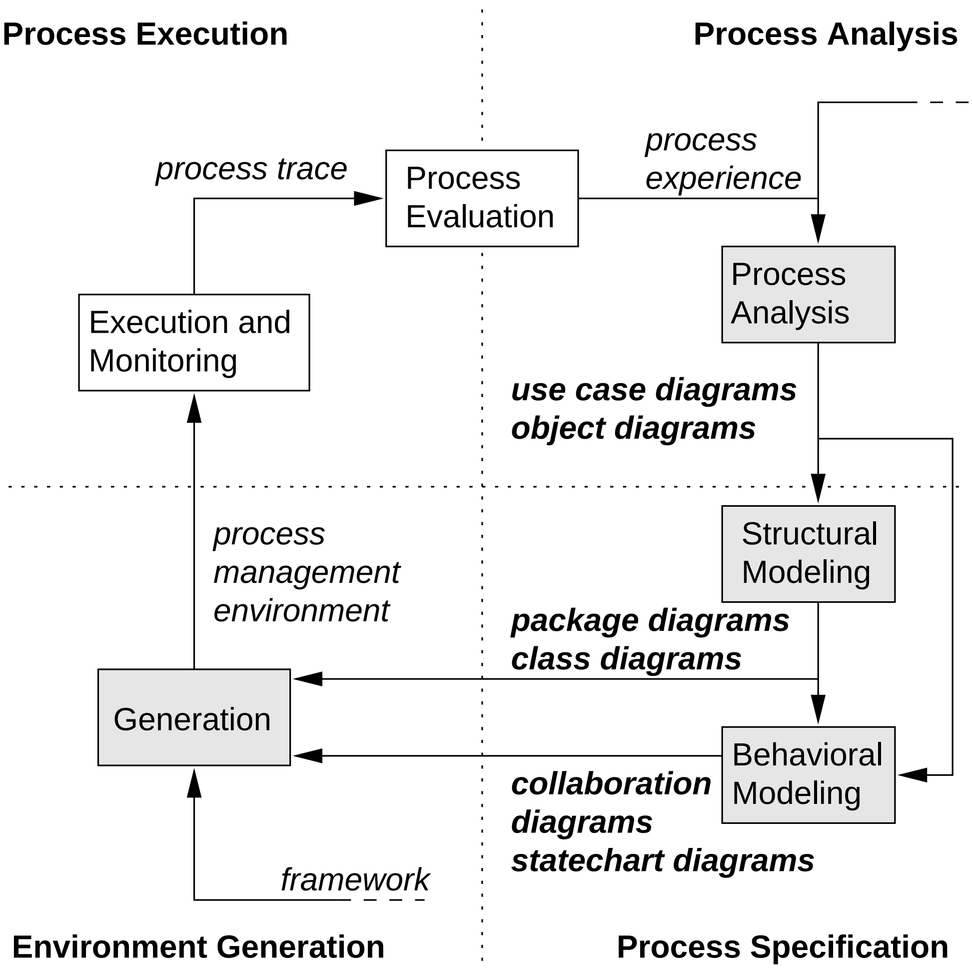

Figure 1: Meta Process

The meta process is depicted in figure 1. It consists of tasks, shown as boxes, and data flows, shown as arrows between tasks. We will use the term *meta task* when we refer to tasks in the meta process to distinguish them from the tasks that are part of the process models we want to construct. We regard the domain of process modeling and execution as being divided into four working areas: process analysis, process specification, environment generation, and process execution. Ideally, the meta process is cyclic and runs through the four areas clockwise. The meta tasks which have to be performed in a certain working area take as input the results of meta tasks of the previous area.

Figure 1 shows the data flow dependencies between meta tasks. The flow of control is not specified here and can be different. For example, the process modeler may realize during process specification that the results of the analysis are incomplete or inconsistent and go back to this working area immediately. The primary aim of this paper is to show how to model processes in UML, not to describe the meta process (i.e. we will be concerned primarily with the products of the meta process). We well therefore not elaborate the control flow relationships in detail. Moreover, we will focus on the working areas of process analysis, process specification and environment generation and discuss the area of process execution very briefly. The meta tasks we will address in detail are shown as grey boxes in figure 1.

The development of a new process model always starts with a *process analysis*. Typically, the input of a first analysis is the process knowledge of humans who performed the process in the past. This knowledge is collected by interviewing people and recorded as *use case diagrams*, where every use case represents a task within the process. Relationships between tasks are expressed by a number of *object diagrams* which are attached to use case diagrams.

The working area of *process specification* deals with the development of the process model from the collected use cases. This can be characterized as elaborating common patterns of task types and their relationships. We distinguish between the meta tasks of structural modeling and behavioral modeling.

Structural modeling is the more basic activity, because before specifying the behavior of model elements it is necessary to specify the elements and their relationships, in other words the structure. Therefore, the structural model is the input for the meta task of behavioral modeling. Since objects are used to represent tasks, task types will obviously be represented by classes inside *class diagrams*, which can be grouped into packages.

As mentioned above, behavioral aspects of the process are recorded informally as notes which are attached to tasks during analysis. *Behavioral modeling* is the meta task of formalizing these notes by specifying the possible transitions between the states of an object and how an event occuring at one task influences other tasks, such as its successors or its predecessors. State transitions are modeled by *statechart diagrams*. Mutual influences between tasks are modeled by *collaboration diagrams*. If no specific behavior is modeled for a task, a predefined standard behavior is assumed.

Within the working area of *environment generation*, the process model has to be transformed into an executable system. Its program code is automatically generated from the UML descriptions of the process. No manual implementation step is necessary here. The resulting system is a prototypical process management environment which implements the process model.

Because one cannot expect to capture all aspects of a real-life process in one step, the first process model and its implementation will almost always be incomplete and sometimes even incorrect. Therefore, one has to monitor and record the execution of a process within the management environment carefully and evaluate the process traces. The meta tasks of *execution and monitoring* and *process evaluation* are part of the working area of *process execution*. Moreover, process evaluation partly belongs to the working area of process analysis, because it serves to discover mismatches between the model and the reality and to identify the parts of the process where further work has to be spent. Typically, this improved process knowledge will result in refining the existing use cases and eventually lead to a new cycle of the meta process.

3 Process Analysis

The aim of process analysis is to capture the process as it is currently performed or as it is in the mind of those people performing it. The resulting process knowlegde is recorded by means of use case diagrams and object diagrams. They are a semi-formal description of process fragments that later will serve as a base for process specification.

The first step within process analysis is to identify the main tasks of the process. For each task, a use case is inserted into the top level use case diagram. If a task can be structured into subtasks, new use case diagrams are added which hold the use cases describing those subtasks. This hierarchy of use case diagrams reflects reality with respect to the delegation

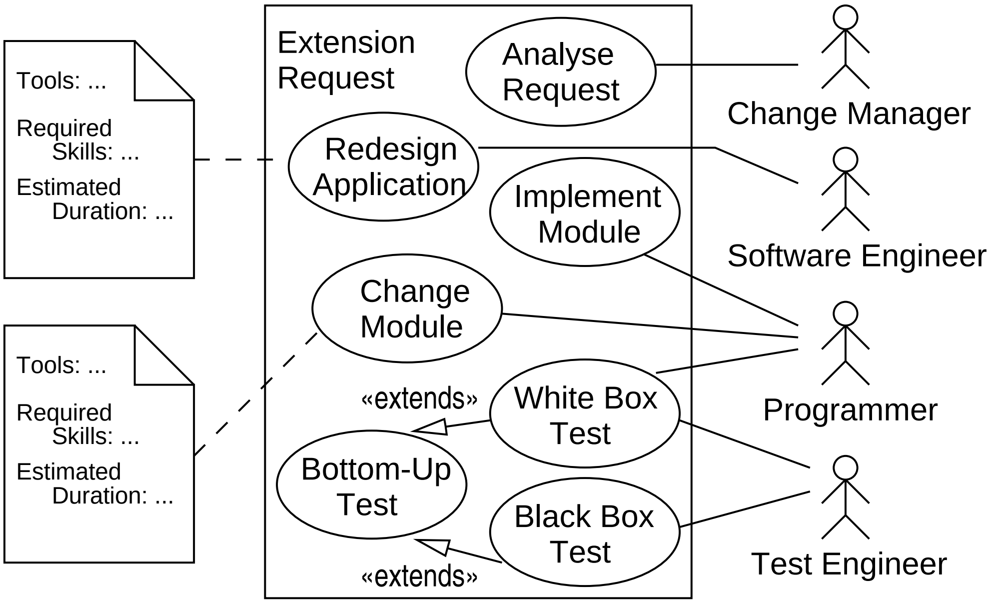

Figure 2: Use Case Diagram

of complex tasks across several levels of organizational hierarchy. Figure 2 shows an example of a top level use case diagram. The use cases within the diagram describe the tasks that may occur while handling a request for extending an existing software system as it might occur during the maintenance phase of a software development process.

The existence of alternative realizations for a task can be expressed by means of extends relationships between an abstract task such as Bottom-Up Test and its concrete realizations such as White Box Test and Black Box Test. Apart from this, the tasks inside a use case diagram are unrelated. Moreover, there may be processes for handling an extension request that do not require all the depicted tasks.

While use case diagrams show which tasks are part of the process, object diagrams are used to describe sequences of task execution, as shown in figure 3. There are two different kinds of objects: tasks and documents. They are distinguished by stereotypes ≪task≫ and ≪doc≫, respectively. Each use case is mapped to a task object in the object diagram. This reflects our view that tasks are products rather than activities. It is important to mention that one of the main problems of object-oriented modeling does not occur in our context: While the identification of objects based on use cases is difficult in general, in our approach there is almost a direct mapping from use cases to tasks in the process model and therefore to objects.

Objects can be associated in four ways. A task may produce or consume a document (stereotypes ≪prod≫ and ≪cons≫). Two tasks can be connected by either a control flow or a feedback flow association. A control flow association (stereotype ≪cflow≫) denotes, that one task is a successor of the other. A feedback flow (stereotype ≪fback≫) means, that due to the results of task execution it might be necessary to repeat the execution of a predecessor task. The direction of control flows and feedback flows is denoted by role names src for the source and trg for the target tasks.

Aspects other than task hierarchy, task sequence and production and consumption of

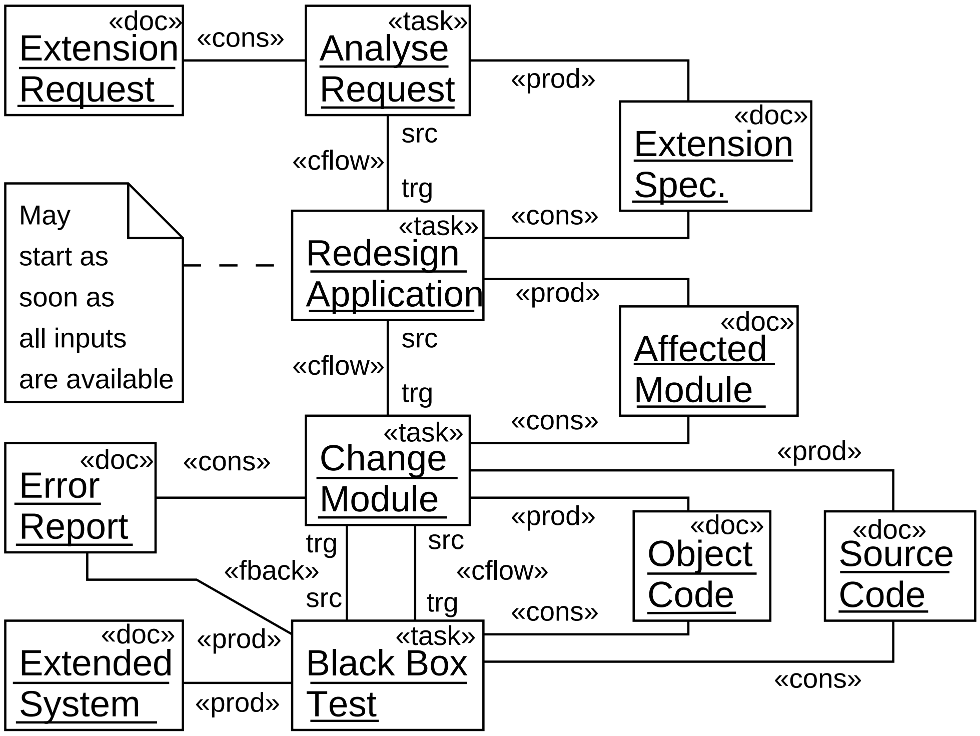

Figure 3: Object Diagram

documents are written down informally as notes which are attached to use cases or objects. Among these aspects are duration of a task, required skill, required tools, etc.

The sample process starts with the Analyse Request task which consumes the document Extension Request and produces the document Extension Specification. There is a control flow association to Redesign Application, which means that this task is a successor of Analyse Request. Redesign Application does not have to wait for its predecessor to terminate. It may start as soon as the input document is available (concurrent engineering). This behavior is expressed by the attached note. The other tasks are connected analogously. Because the object diagram shows a concrete execution of the process, it contains one of the realizations of the abstract task Bottom-Up Test, namely Black Box Test. There is a feedback flow from Black Box Test to Change Module expressing that Change Module must be reactivated if an error is discovered during the test. In this case, the results of the test are passed as an error report.

Object diagrams are attached to use case diagrams and might already include a first step of abstration, where objects do not represent tasks but sets of tasks. The level of abstraction varies with the method we use to gather knowlege. If we record process traces by monitoring, we will always get concrete information. If we talk to people we will get a mixture of concrete information, e.g. examples of how processes were performed in the past, and abstract information, e.g. about the rules that guided people through the process.

Typically, there will be several object diagrams attached to one use case diagram. Different diagrams either describe disjoint parts of the process or alternative realizations. They may even contain redundancies for the goal of process analysis is to collect as much information as possible without filtering it at this stage.

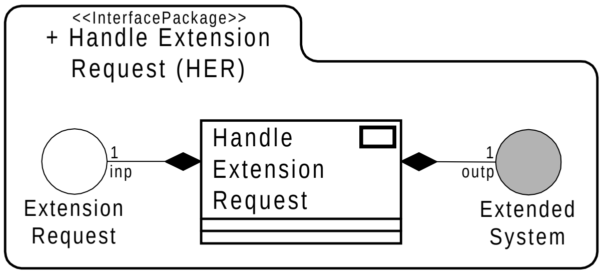

Figure 4: Task Interface

4 Process Specification

Specifying an executable process model means to formally define its structure and its behavior. To increase model understandability and to decrease model maintainance effort, process specification should be supported on a very high level of abstraction. In contrast to other approaches introducing a special purpose process modeling language (EPOS [9], JIL [21], MADAM [19]) this section will demonstrate how process models can be adequately defined in a standardized object-oriented modeling language, the UML. The transition from analysis to specification will be discussed at the end of this section.

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. A process' structure consists of task, parameter and realization classes and their various interdependencies. Consequently, we use UML's *class diagrams* to model this structure. Since in contrast to process analysis executable models are specified, the meta model for process specifications is more detailed.

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 project manager at process enactment time.

Figure 4 shows the interface of the task class Handle Extension Request. We make explicit use of UML's *stereotype* concept and introduce stereotypes for task classes (symbolized by a simple rectangle), and input and output parameter classes (symbolized by a white and grey circle, respectively).

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 all detail later on.

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, these realization classes are not part of the interface package. Rather an individual package is introduced for every realization class. Realization classes abstract from complex schematic realizing subprocess definitions as shown in figure 5. These allow the *composition* of other task classes through the following associations which were introduced in section 3: *Control flow*

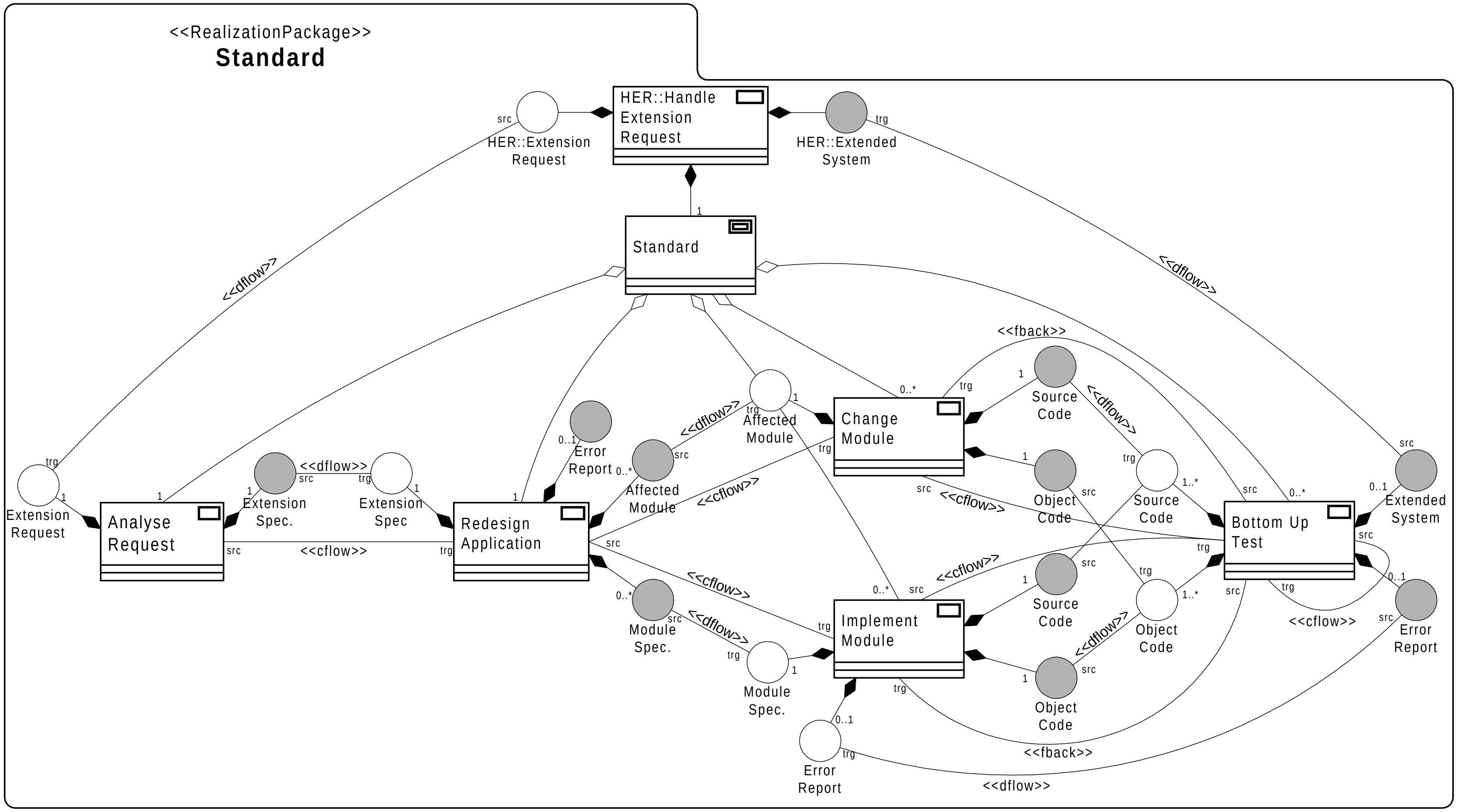

Figure 5: Task Realization

associations introduce a temporal 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 rolenames src and trg. To make these associations distinguishable, we introduce a set of 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 as explained in section 3). 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.

Task interfaces and realizations are stored in separate *packages*. Packages in UML are utilized to group closely related modeling elements together, to offer a separate name space and to define visibilities. To distinguish the two types of packages we define the stereotypes ≪InterfacePackage≫ and ≪RealizationPackage≫. Figure 4 shows a sample *interface package*, while in figure 5 a sample *realization package* is displayed. The visibility of interface packages is inherently public (+) because they offer a task's contract for external use in realization packages. In contrast, the visibility of realization packages is inherently private (-) because process models using the exported task class are not concerned with the internals of a particular realization package.

Figure 6 gives an example of how packages should be used to structure the process model. An interface package may contain several realization packages, namely one for each realization class. For example, a bottom-up test may be carried out as a white-box or a black-box test, which results in two realization packages being contained in the interface package Bottom-Up Test.

All task classes needed to model a realizing subprocess for a complex task class may be contained in the appropriate realization package. The Develop Test Cases packages are nested in the two shown realization packages as the development of test cases differs for

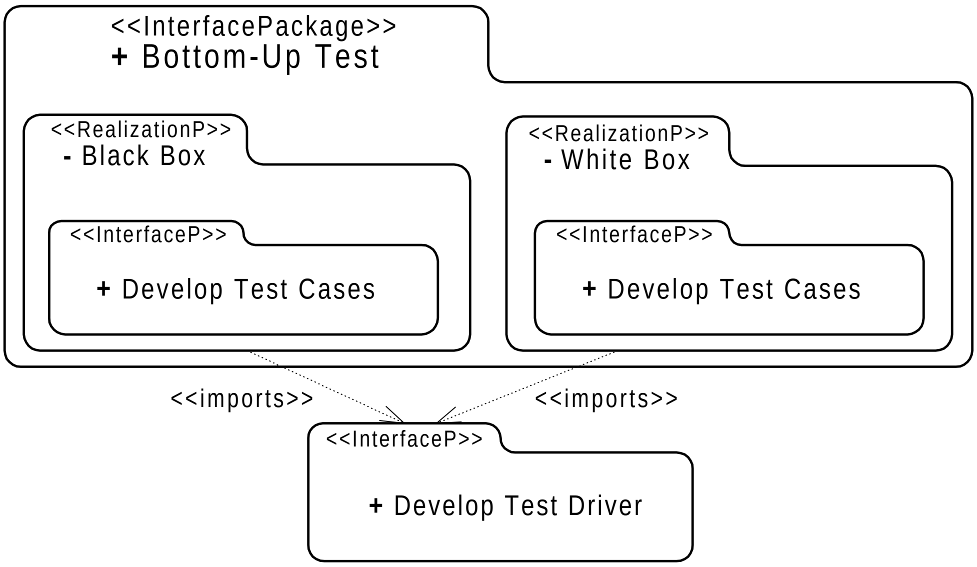

Figure 6: Usage of Packages

black box and white box testing. However, there will exist task classes that can be used in varying contexts to model a subprocess. These should be modeled as stand-alone interface packages to allow several realization packages to import their offered contract. For example, the processes for developing test drivers for black box and white box testing can be the same. A dependency of stereotype ≪imports≫ marks the set of used interface packages for each realization package.

The advantages of using UML packages in the afore mentioned way are the following ones: They allow for the modular modeling of processes so that complex processes are separated into manageable fragments. In addition, reusable process fragments can be identified since each stand-alone interface package is subject to reuse.

4.2 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* (cf. figure 7). 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). Figure 7 shows all state-changing methods but only gives 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 7 is sufficient to adequately model a process.

The state diagram allows tasks to be currently defined (InDefinition), to be waiting for activation, to be active, suspended or replanned. Every task can terminate in one of two

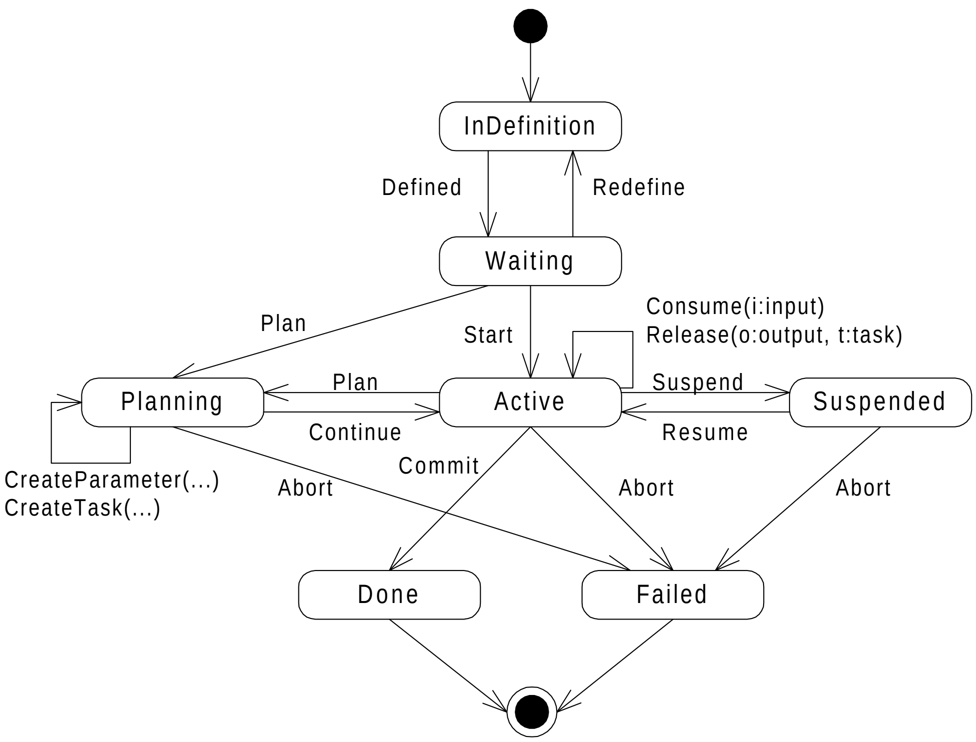

Figure 7: Generic State Diagram

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 preceeding 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 or abortion of a task leads to the suspension or abortion of all subordinate tasks[1]. This standard behavior can be enhanced by the process modeler in multiple ways.

One way to model a task class' specific behavior is to define *conditions* for the transitions of the state diagram. These conditions provide the means to influence the way a task net is executed. Different development policies like concurrent and sequential engineering can be realized or certain states can be eliminated from the state diagram by disabling its incoming transitions. Conditions must be formulated in the *Object Constraint Language* (OCL) which is part of the UML. The OCL offers a rich set of constructs to model conditions.

A sample enhanced state diagram for the Redesign Application task class is shown in figure 8. It contains a condition for the start transition of the owning task class which may be executed only when all inputs are available (the standard behavior does not demand this). Therefore, all input parameters of the owning task are collected and for each of them a value of true is required for the InputAvailable attribute.

Since the start transition for tasks belonging to the class Redesign Application can be executed only when all inputs are available, we can now automize the consumption of its

[1] The complete standard behavior is defined in [12].

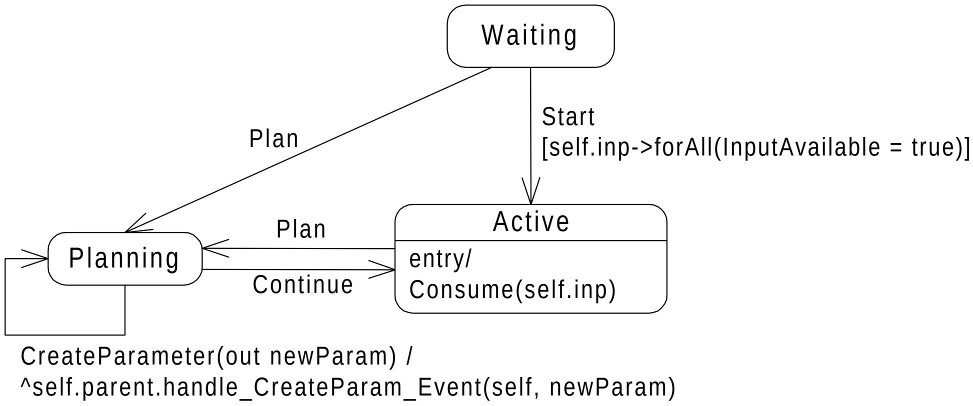

Figure 8: Specific State Diagram

inputs. 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 certain state. The sample state diagram of figure 8 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. This leads to a clean separation of proactive (automated execution of methods) and reactive (event handling) behavior.

Figure 8 shows how the set of event-receiving tasks can be restricted through send-clauses at specific transitions. The sample state diagram shown in figure 8 restricts the target set of the CreateParameter event to the task's parent. 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 which is explained next.

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++ 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 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 class.

Complex task net transformations can be specified through UML's *collaboration diagrams* which are used to define pre- and postconditions of an event handling method. Figure 9 gives an example of how a CreateParameter event can be handled through this mechanism. The precondition requires the new parameter to be of type Affected Module and its attachment to a task of type Redesign Application. The precondition plays the role of a task net pattern that

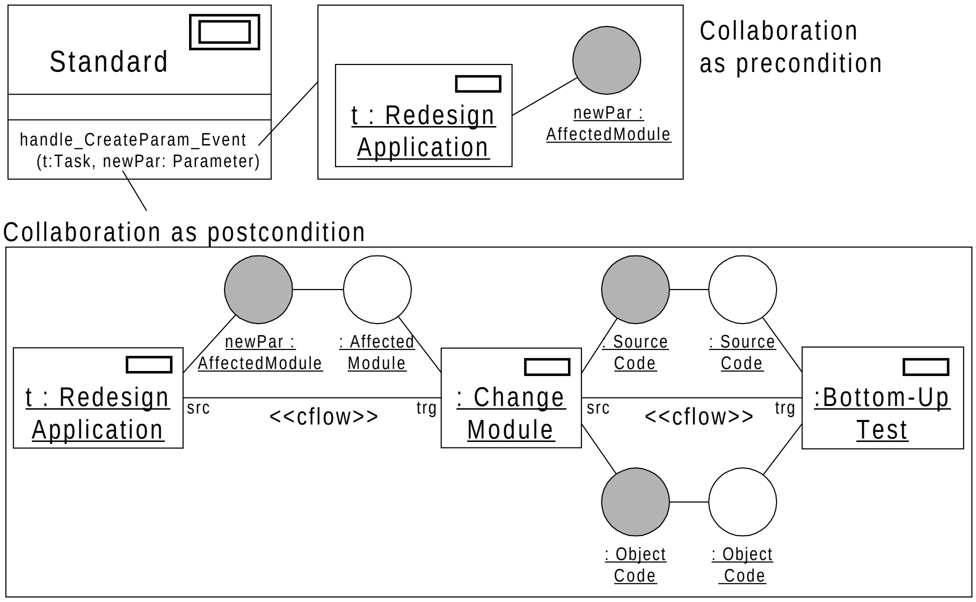

Figure 9: Complex Task Net Transformation

is searched for in the current task net. The postcondition specifies the transformation being executed on this task net. As for the extension request handling process it is well known that every affected module has to be changed and the resulting module implementation has to be tested. We can implant this knowledge into our process model.

The collaboration diagram used as a postcondition requires new tasks of type Change Module and Bottom Up Test to be created and to be connected via control flow links. Their respective in- and output parameters are also created and connected via data flow links. Since the used collaboration diagrams are nothing but fragments of instance level task nets, we can provide support for round trip engineering of process models.

This example only uses the static properties of collaboration diagrams. However, we equally support their *dynamic properties* like conditions and sequences of method calls. Using the dynamic features would for example allow for the predefinition of controlled reactions to the occurrence of feedback. Figure 10 shows a sample collaboration diagram for handling feedback between the test task and a redesign application task which may occur when design errors become apparent during the test. The event handler initiates the suspension of the feedback's source, enforces the release of the faulty module specification to be revoked and suspends the implement module task working with the faulty specification.

Specifying the behavior through state diagram adaptations and the definition of custom event handlers can become a tedious task. However, 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 or inputs that are either required for activiation or may still arrive when the task is already active. 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.

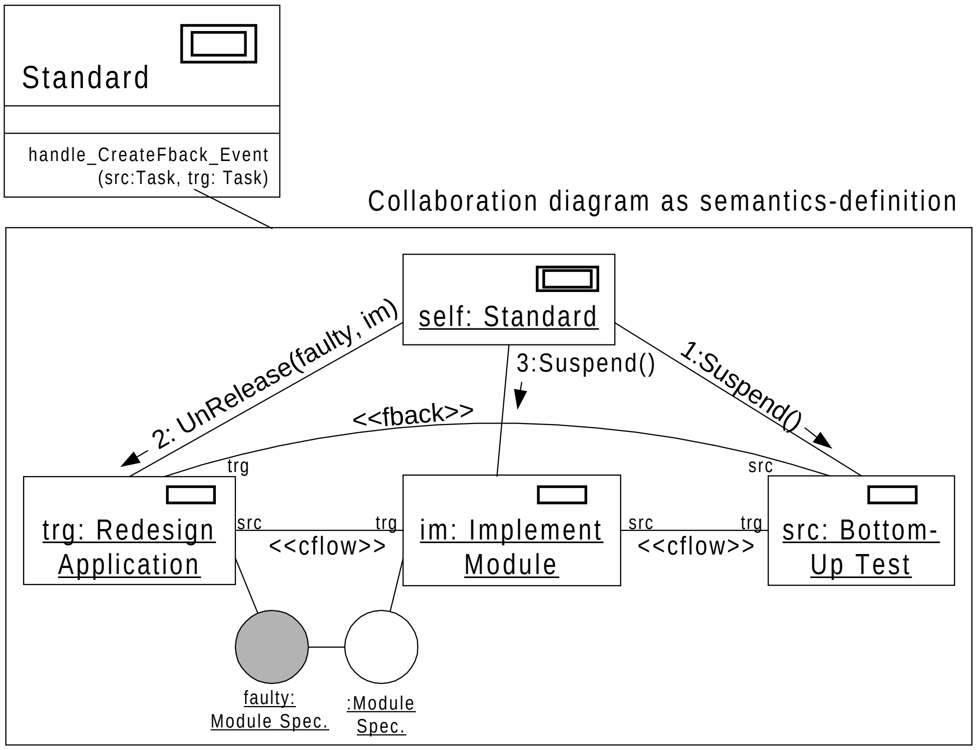

Figure 10: Handling of Feedback

4.3 From Analysis to Specification

The transition from analysis to specification requires a lot of process knowledge and creativity. However, some heuristic rules can be offered at least for structural modeling (behavior is described only informally during process analysis).

Every use case is a candidate for a task class. A use case diagram is mapped into a complex realization. For use cases that are not refined by use case diagrams, atomic realizations are defined. In this way, the hierarchy of use cases is mapped 1:1 into a composition hierarchy of task classes.

The composition of a complex realization is obtained by analysing object diagrams. First, we have to determine the classes of task and parameter objects. This identification problem can be solved more easily if task objects have been designated as instances of use cases during process analysis. Then, we may infer associations between these objects and determine cardinalities that are consistent with the object diagrams collected during process analysis.

This briefly sketched method of specifying a process model according to the results of process analysis is very simplified. Deriving the process model's structure on the type level from the object diagrams is in fact a difficult task as the right abstractions have to be found. There might be redundant use cases that can be modeled as one task class. Object diagrams describing one use case may not be consistent with each other so that conflicts have to be solved. In general, there may be no direct mapping from the hierarchy defined in the use case diagrams to the hierarchy of class diagrams, especially when use cases are defined redundantly.

5 Environment Generation

The 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.

We employ the DYNAMITE environment [7] to support process enactment. DYNAMITE has in turn been constructed with the help of PROGRES [20], an environment for executable programmed graph rewriting systems. To this end, we have specified the process meta model for dynamic task nets as a programmed graph rewriting system [8, 12]. Node types, edge types, and attributes are defined in a graph schema; graph transformations are specified by graph rewrite rules and procedures of these. The choice of the specification language is consistent with our view of processes as objects; all operations on task nets are specified by graph rewrite rules in a uniform way.

This specification provides *base operations* for editing task nets, performing state transitions, flow of data, etc. It is concerned with "untyped" task nets (all tasks are instantiated from a single type) with some standard behavior that has been explained briefly in the last section. The 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.

A process specification in UML is transformed into a corresponding specification in PROGRES which extends the base specification. UML class diagrams are transformed into a graph schema defining domain-specific task and relation types. State transitions are transformed into procedures calling the base operations. Graphical definitions of pre- and postconditions of event handlers are transformed into graph rewrite rules. Finally, collaboration diagrams are transformed into procedures which perform method calls.

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. Low-level process programming is avoided completely.

6 Related Work

Throughout this paper, we have been concerned with process modeling <u>in</u> UML. We have not discussed process modeling <u>for</u> UML, i.e., methods for applying UML to software development (e.g., the Objectory process [10]). 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 (see conclusion).

Process modeling in UML is often performed with the help of activity diagrams [23, 11]. However, we have already pointed out earlier that usage of activity diagrams does not take process dynamics properly into account.

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+ [18], 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 [3].

To conclude this section, let us briefly 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 [22] or Petri nets [2]. 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 [9]. 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.

7 Conclusion

We have applied a standard object-oriented modeling language (UML) as a "front end" to a process management environment. In this way, we exploit the high-level modeling features of UML, using a wide-spread standard notation, and make UML process models executable by a management environment for dynamic task nets. In fact, UML offers a rich variety of modeling elements for process modeling. Use case, object, class, statechart, and collaboration diagrams are combined into expressive process models which can be transformed into an executable form. By enriching the facilities of the underlying process management environment and providing these facilities through stereotypes and properties at the UML level, process models can be described at a very high level of abstraction so that the need for low-level programming is eliminated.

In our work, we intend to support the modeling and execution of dynamic software processes, taking e.g. product evolution, feedback, and concurrent engineering into account. In addition to software processes, we have been studying other domains as well (mechanical engineering [24] and — only recently — chemical engineering). Our experiences in all of these domains have shown that processes exhibit quite similar characteristics, and that dynamic task nets are quite universal with respect to their application domain.

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 [12, 19]. We have also implemented a modeling environment supporting the creation of MADAM models and their translation into a programmed graph rewriting system. The implementation has been created with the help of the PROGRES environment. For the UML-based modeling environment, however, we are using a commercial CASE tool instead.

The process management environment (DYNAMITE) is currently being re-designed and -implemented. In particular, we are going to enhance the user interface with the help of a commercial tool-kit, and we have selected Java as our new implementation language (formerly, the implementation was written in 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. [4], pages 279–292.

[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] G. Engels and L. Groenewegen. SOCCA: Specifications of coordinated and cooperative activities. In Finkelstein et al. [4], pages 71–102.

[4] A. Finkelstein, J. Kramer, and B. Nuseibeh, editors. *Software Process Modelling and Technology.* Research Studies Press (John Wiley & Sons), Chichester, UK, 1994.

[5] A. Fuggetta and A. Wolf, editors. *Software Process*, volume 4 of *Trends in Software.* John Wiley & Sons, New York, 1996.

[6] 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.

[7] 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.

[8] P. Heimann, C.-A. Krapp, B. Westfechtel, and G. Joeris. Graph-based software process management. *International Journal of Software Engineering and Knowledge Engineering*, 7(4):431–455, Dec. 1997.

[9] M. L. Jaccheri and R. Conradi. Techniques for process model evolution in EPOS. *IEEE Transactions on Software Engineering*, 19(12):1145–1156, Dec. 1993.

[10] I. Jacobson. *Object-Oriented Software Engineering: A Use-Case Driven Approach.* Addison-Wesley, Reading, MA, 1994.

[11] 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.

[12] C.-A. Krapp. *An Adaptable Environment for the Management of Development Processes.* Number 22 in Aachener Beiträge zur Informatik. Augustinus Buchhandlung, Aachen, Germany, 1998.

[13] P. Lawrence, editor. *Workflow Handbook.* John Wiley & Sons, Chichester, UK, 1997.

[14] L. Osterweil. Software processes are software too. In *Proc. ICSE'9*, pages 2–13, Monterey, CA, Mar. 1987.

[15] Rational. *UML Notation Guide*, 1.1 edition, Sept. 1997. http://www.rational.com/uml.

[16] Rational. *UML Semantics*, 1.1 edition, Sept. 1997. http://www.rational.com/uml.

[17] R. Reddy et al. Computer support for concurrent engineering. *IEEE Computer*, 26(1):12–16, Jan. 1993.

[18] 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.

[19] A. Schleicher. High-level modeling of development processes. In *First Int. Conf. on Process Modelling*, Cottbus, Germany, Feb. 1999. Accepted for publication.

[20] 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.

[21] 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.

[22] 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.

[23] G. Versteegen. Objektorientierte Geschäftsprozeßmodellierung mit der UML: die Innovator Business Workbench. *OBJEKTspektrum*, (1):62–67, Jan. 1998.

[24] B. Westfechtel. Integrated product and process management for engineering design applications. *Integrated Computer-Aided Engineering*, 3(1):20–35, Jan. 1996.