An Environment for Managing Software Development Processes

Peter Heimann, Carl-Arndt Krapp, Bernhard Westfechtel
Lehrstuhl für Informatik III, RWTH Aachen
Ahornstr. 55, D-52074 Aachen, Germany
[peter|krapp|bernhard]@i3.informatik.rwth-aachen.de

Abstract

Considerable efforts have been undertaken to construct environments supporting dynamic software processes. The road to success neither takes us into the runtime stack of the process engine (rule-based approaches), nor may we tolerate the straight-jacket of a static Petri net. We propose an environment centered around dynamic task nets, which provide project managers with all details required to plan, analyse, and monitor software projects, taking into account changes to the product structure, feedbacks, simultaneous engineering, and replanning of resource assignments. Developers benefit from the management of dynamic work contexts, including control over incoming versions of input documents and intermediate releases of output documents. The environment is based on a formal specification as a graph rewriting system, which allows for describing editing, analysis, and enactment of task nets in a uniform formal framework. By generating code from the specification, we avoid the cumbersome and error-prone task of translating complex graph transformations manually into a low-level programming language.

1. Introduction

Process management is a challenging discipline within the software engineering community [4]. It focuses on the support of the software development process in order to shorten development time and to increase product quality. A lot of work has been invested to find suitable models to represent and enact the process. Development environments have been extended by process descriptions, supporting software engineers in their everyday work. We believe that many of the proposed approaches could not find their way into application because of several shortcomings: The models were unable to deal with dynamic situations within the development process, end user support was concentrated on the software engineer and neglected the project manager, and some approaches failed to provide a representation managers and developers could easily understand.

Development processes are highly *dynamic*. Reasons for change include initially incomplete or changing requirements, introduction of new tools, changes in the developer team, new priorities and deadlines, reuse of partial results from other projects, and last but not least correction of mistakes in plans and technical documents. The development process is therefore not completely plannable in advance. Though every project is different, a company wants to continually improve their processes. Not only documents shall be reused, but also successful patterns for their creation shall be identified and used for the next similar task. To make this possible, not only the document versions have to be archived, but also the tasks and their relations.

We have built an environment to support process management and have put the emphasis on handling continuous change. This environment contains tools for three user roles:

- To *developers*, an agenda of assigned tasks and a work context for the current task is presented. This context changes not only by actions of the developer, but also by external influences: new versions of input documents arrive, formerly released document versions are retracted, etc. When a completed task has to be reactivated, the old work context is reconstructed and extended by error reports or change requests.
- *Project managers* see a representation of all tasks and their relations. They initially plan the project they are responsible for and modify the plan during enactment, assign tasks to developers and arrange resource usage. The manager watches the changes to the project status, as developers create documents and complete tasks. New tasks are introduced depending on interme-

diate results, either automatically according to predefined rules or manually. When a problem situation arises, the manager may have to create new tasks to identify the error, then reactivate completed tasks or create new ones to correct it.

- *Process modelers* define the application domain specific parts of the model and environment on a high logical level and thus create the boundaries for the managers' and developers' work. Without delving into the details of the process formalism, they can select model components from a library. They extend the process model schema, define task patterns and preconditions for status changes, and determine the general way changes and errors are handled. The library contains elements to achieve the usually required behaviors, new elements can be added upon need.

We have proposed the *DYNAMITE* model for planning, analyzing, and executing evolving development processes using hierarchical task nets. The model allows execution of incomplete process models, modifications on the fly, intertwined planning, modification, and enactment of process descriptions ensuring model consistency. While [11] presented the formal foundation, this paper focuses on describing an adaptable environment that realizes the DYNAMITE model and can handle dynamic situations for developers and managers.

The remainder of this paper consists of three parts. First, we describe the advantages of our approach compared to other process formalisms. Then, the process management system is presented from the views of the three user roles. Finally, we concentrate on the specification and the construction of the environment using graph rewriting systems, which are a natural way to seamlessly integrate editing and execution of task nets.

2. Related Work

The conceptual distinction between three different roles in the software process is not new. [6] distinguishes between a process definition domain, a process definition enactment domain, and a process performance domain. This corresponds with our distinction between a process modeler, who defines process enactment patterns, a process manager, who takes such patterns to instantiate the development process, and the technical developers, who carry out the defined process. Along the three levels, we compare our approach to some rule-based and some net-based approaches for process modeling and enactment.

On the *process definition level*, process modelers characterize processes or fragments of processes in some languages in terms of how they could or should be performed. Rule based approaches like ALF [2], MARVEL [12] and MERLIN [13] use a Prolog-like notation. While the execution semantics are defined by the rule inference machine, the execution order is derived from the product structure, which is defined in an ER-like notation.

SPADE [1], Process Weaver [7] and LEU [5] define processes by Petri nets. Execution semantics are based on the Petri net semantics and can be adapted by the definition of pre- and postconditions. Other net based approaches like EPOS [3] and Teamware [21] define data- and control flow dependencies on a type level, which constrain the structure of net instances. The net is interpreted by a process engine.

We use the specification language PROGRES [18] to define how a certain class of attributed graphs can be built and modified. PROGRES serves as a DDL known from database management systems which allows us to define the structure and semantics of the persistent data structure representing dynamic task nets. In contrast to other approaches, where execution semantics are implicitly defined by the execution semantics of the underlying language, we rather define the semantics of task nets explicitly by means of programmed graph transformations. The graph schema restricts the evolving graph structure. A tool environment for dynamic task nets can be generated from a PROGRES specification [19].

On the *process definition enactment level*, process definitions are used to instantiate the process. In rule based approaches, the inference machine is working on its fact and rule base. The structure of task nets is constructed implicitly by forward and backward chaining [12, 3, 2]. Changes in the product structure imply changes in the process structure. However, as noted in [3], it is crucial to support human intervention. Automatically constructed task nets fail to meet this requirement. In particular, project managers must be able to operate on task nets, which may still be generated partly automatically.

Net based approaches suffer from the essentially static structure of its process definition. The topology of the process description cannot be modified at all [14], or only at certain points, implying that not all corrective modifications on the enacted process definition are possible [1, 5, 7, 3]. There is usually no equivalent to the task nets found in rule-based systems, which are constructed at runtime [12, 2, 3]. Modifications to the net topology are considered as serious interventions which affect the process definition [1, 5, 7].

We present new aspects which we have not seen in the literature so far. Our approach meets the requirements formulated by [6] that a human process manager is necessary, whose job is to monitor process definition enactment and process performance, and to take corrective actions when necessary. The process manager is provided with several views on an enacted task net, which in turn is a user friendly representation of a graph. By means of the defined graph operations, intertwined editing and enacting of task nets is realized. A manager may suspend arbitrary tasks, introduce feedbacks and analyze its impact, or replace some complex time consuming tasks by simpler ones.

On the *process performance level*, rule based approaches support technical developers actively by the provision of a task list or passively by offering information on demand about things to do. For well defined process chunks without human interaction, approaches like [2, 12] offer automatic process performance with sophisticated tool integration [9]. Feedbacks are handled rather pessimisticly by restarting all potentially relevant development tasks.

Net based approaches offer similar user interfaces to developers, but they tend to only offer active and automatic process performance [1, 7]. Feedbacks are represented by cycles in the topology of the net. Although alternatives in processing feedbacks can be accommodated, assessing their consequences on on-going development tasks would require instance-level nets.

The DYNAMITE model can offer all different degrees of user guidance in the same development process. First, a process manager may plan a rather coarse grained process model, where a developer has any freedom to proceed. After more information is available about the process, the manager may detail the model during project performance, which subsequently constrains the developer in his doing. Feedback handling is more sophisticated, since the process manager may either analyze each feedback manually, or use predefined patterns specified on the process definition level. In particular, instance-level task nets are used to assess the consequences of feedbacks on on-going development activities.

3. Process Definition Enactment Level

The tools for the project manager are centered around graphical representations of task nets, which look similar to PERT charts and other traditional project management methods. The manager can arbitrarily switch between different views, where he can observe and influence the project's execution. All views are derived from a single internal database for the whole project, and all change commands modify this database, so that consistency of the views is guaranteed.

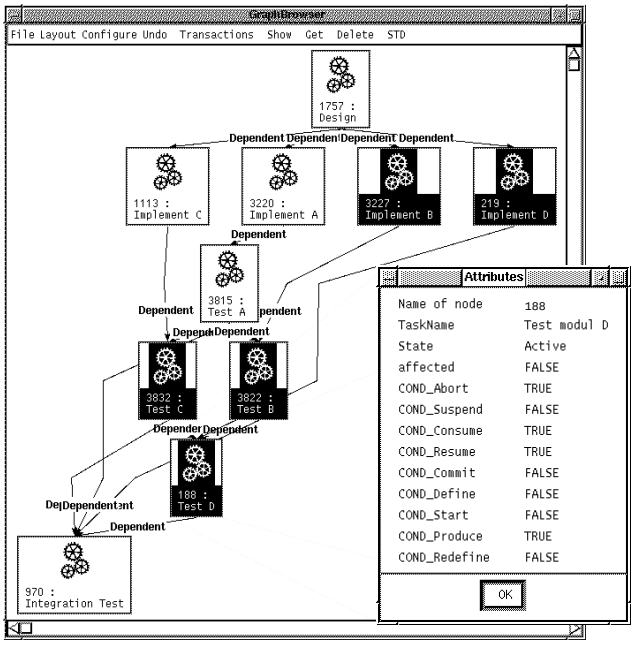

Figure 1. Control flow view

We explain the tools by means of a simple software subsystem development example. The subsystem consists of modules `A`,...,`D`, where `B` and `C` import from `A`, and `D` imports from `B` and `C`. Subsystem design should be followed by concurrent implementation of all modules. Subsequently, modules should be tested in a bottom-up fashion. Finally, an integration test should reveal system's performance and robustness.

Figure 1 shows the *control flow view* of the development project. Each task is represented by a node, and the tasks are connected by various relations. Each task can be atomic, or refined by a task subnet. Figure 1 displays the control flow relations (`Dependent`), which form a DAG and determine task execution order. Additionally, along the control flows data flows define the exchange of document versions between tasks.

Each task possesses a set of *attributes*, among them the `State` attribute. Usually, the state attribute gets modified because of technical developer's actions. For example, it changes to the value `Active` (indicated by a black icon) when a developer begins to work on the task. Condition attributes indicate whether a corresponding state transition is currently possible. Evaluation rules for these attributes are specified by the process modeler.

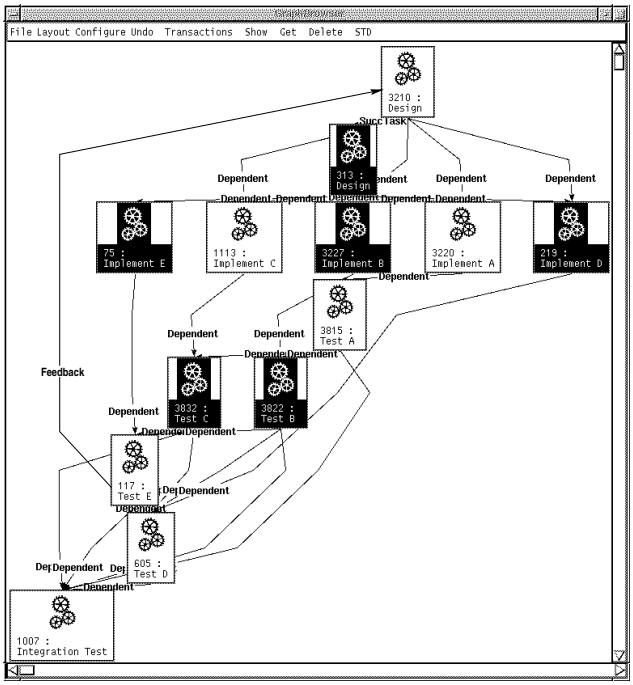

Figure 2. Modified task net

The DYNAMITE approach allows for execution of incomplete task nets which can be *extended dynamically*. Tasks have to be fully defined only when their execution shall be started. The sample project in figure 1 has begun with only the `Design` and `IntegrationTest` tasks. After the coarse design had been created, the implementation and test tasks and their relations have been added accordingly. Some time later, the situation shown in figure 1 has been reached, with developers working on the implementation of modules `B` and `D` and several module tests in progress.

In addition, *unforeseen circumstances* like technical problems, running out of time and money, etc., may require changes to task subnets currently being carried out. Let us assume that the developer responsible for the implementation task of module `D` recognizes that an architecture transformation would be worthwhile. He informs the project manager, who decides to redesign the software system. Therefore a feedback from the implementation task back to the design task (`Feedback`) is added. In the DYNAMITE approach, a new version of the design task is introduced (cf. node 313 in fig. 2), because we want to be able to later trace what has happened. This would not be possible by just reactivating the old task. In our example, the new software architecture contains a new module `E`. While other active tasks continue to run unaffected, an implementation task and a test task for the new module are inserted.

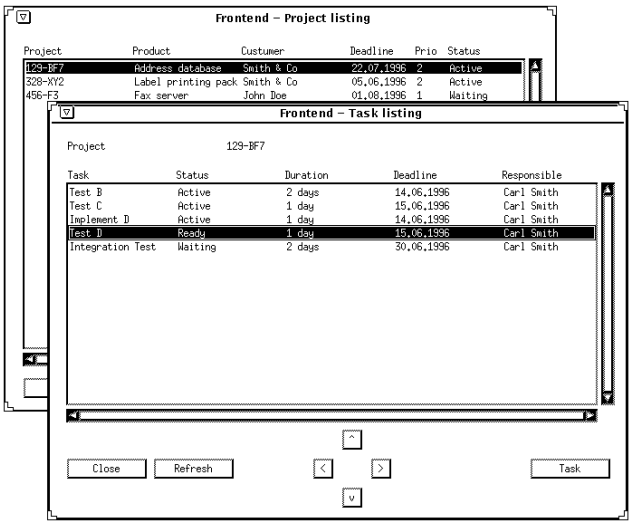

Figure 3. Agenda for developer

Other views, not shown here, are available to the project manager as well. A *feedback view* helps to estimate the consequences of task reactivations. It displays all tasks in the (transitive) control flow chain of a selected task. A *data flow view* presents for a task and its input/output parameters all tasks and parameters which are related by control flows and data flows, respectively. Furthermore, a hierarchical view presents all tasks found in one hierarchy level.

4. Process Performance Level

Technical developers can see, but not directly change cutouts of task nets. The main views a developer uses to interact with the process management system are the agenda and the work context.

The *agenda* (fig. 3) displays tasks which are assigned to the developer and which have not yet been fulfilled. Tasks are grouped according to the projects they belong to. Among these, the developer selects the task he wants to work on next. A task which may be started is shown in state `Ready`. Preconditions for this are defined by the process modeler (see section 5), they might e.g. require that all input documents are available. To announce to the system that he will work on the task, the developer inititates a transition into the state `Active`.

The *work context* for a task (fig. 4) consists of four groups of documents. Inputs are produced by preceding tasks in the task net, outputs are the results to be produced and passed to successor tasks. Guidelines are documents that are valid for several tasks and which seldom change, like norms and coding rules. Auxiliary

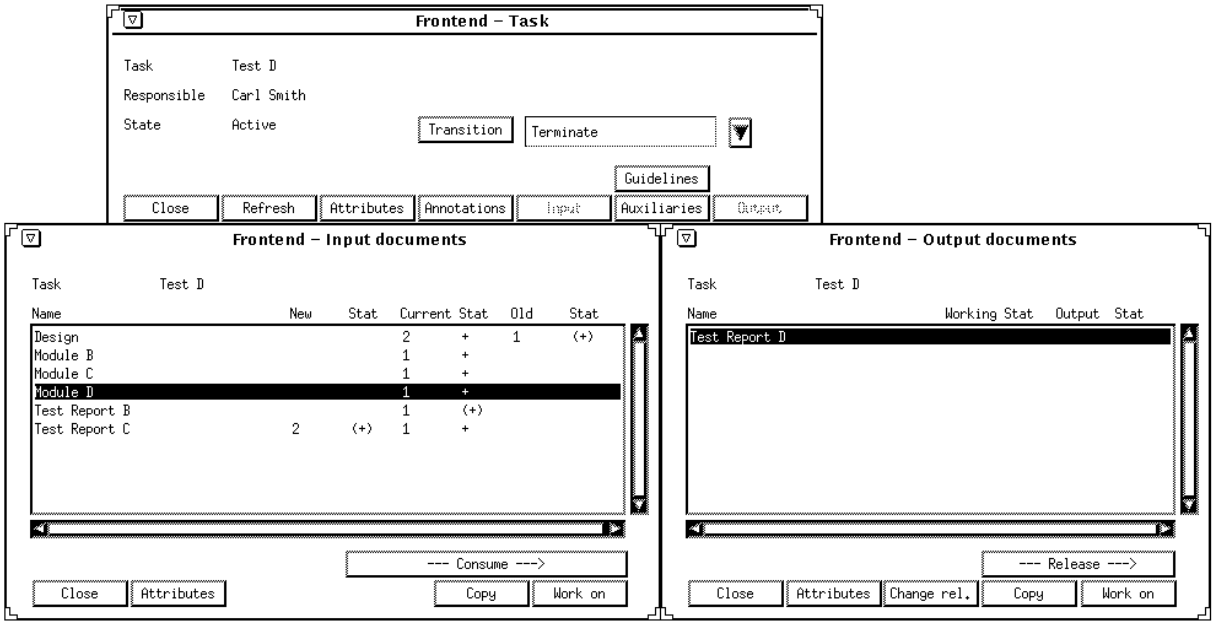

Figure 4. Work context

documents are local to the task, they belong to the private workspace for the developer.

To view or modify the contents of document versions, the developer starts suitable *external tools*, but the environmet shields him from all technical details of doing so. Upon start of the tool, a document version is checked out to the local host, and the tool runs on a local copy of the version. When the developer finishes using the tool, a changed version is checked in to the repository. Inputs and guidelines, however, can not be modified.

A subset of the *version history* is displayed in the work context. For input documents, this encompasses a new and old version besides the current one. A new version that arrives after the task has been started is shown in the leftmost of the three version number columns of the input documents' window (see e.g. `Test Report C` in the `Input documents` window of fig. 4). The developer explicitly tells the system (by clicking on the `Consume` button) when he wants to replace his current version with the new one. The new version then becomes the current version, the former current version can be accessed as the old version. The visible effect thus is that the column contents are shifted one step to the right.

On the output side, the current version of a document can be *released* to all or a selection of successor tasks, where they are immediately displayed as new input versions. Symbols in the `Stat` column show the condensed release information for the version: `-` means an as yet unreleased version, a version with `(+)` has been released to selected successor tasks, and `+` to all successors.

A released version is frozen in order to ensure traceability. Starting a tool to change it triggers the automatic creation of a new work version. Should the developer discover an error later on, or should new input document versions make the output inconsistent, he can retract the release, again totally or for selected successor tasks.

The work context can thus change during execution of a task: developers in predecessor tasks produce new document versions, and the project manager can even add new input documents. By the explicit `Consume` operation, the developer controls when he wants to incorporate these external changes into his work. By explicit releases and partial releases, simultaneous engineering is supported in a controlled way.

Task *state transitions* can be performed by the developers, if the preconditions hold. The most important transitions are temporary suspension in case of problems, and termination. Conditions for the successful termination of a task as defined by the process modeler can include:

- For all output documents, at least one version has been produced.
- The most recent versions of all input documents have been consumed.

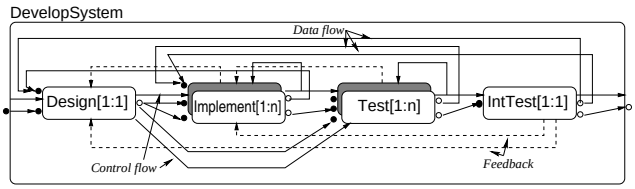

Figure 5. Realization type definition

- All predecessor tasks have been terminated, so they cannot produce new input versions.

In this way, the system prevents premature termination of tasks. A terminated task disappears from the agenda. It reappears when the project manager creates a new version of the task.

5. Process definition level

While in the former subsection we were concerned with the views and tools at the enactment and performance level of the process management system, we are now switching to another logical level of modeling. The *process definition level* is used to adapt the management system to a specific scenario in order to enhance the reusability of the system and its encapsulated knowledge. We describe the adaptable parameters and how a process modeler is supported in defining them.

When modeling the software process, in many cases *a priori knowledge* is available. We may identify types of tasks, which have common input and output parameters, and may impose an order between types of tasks and parameters (cf. fig. 5). For each task type, several realization types can be defined. They either describe a suitable structure of a refining task net, or they describe a tool which can be used in order to solve tasks of this type. The type information can be used by the project manager for instantiating and executing task nets.

In our approach, execution semantics of dynamic task nets are based on a state transition diagram. Transitions are either performed explicitly by process managers and developers, or implicitly by event-trigger rules (see below). For each task type, the process modeler can adjust the *transition conditions*. Tables 1 and 2 illustrate the effects of different conditions for the `IntTEST` and `TEST` task type. It shows which states predecessor tasks might have when the task of a corresponding type has a certain state. While test tasks might start their work before all predecessors are done, the integration task must wait until all modules have been successfully tested. A predefined library of different condition sets is provided to a process modeler, where he is not bothered with formulating complex boolean expressions.

Table 1. Predecessors' state of `IntTEST` task

	predecessors' state						
IntTest	InD	Wai	Pln	Act	Spd	Dne	Fai
InDefinition	x	x	x	x	x	x	x
Waiting	x	x	x	x	x	x	–
Planning	–	–	x	x	x	x	–
Active	–	–	–	–	–	x	–
Suspended	–	–	–	–	–	x	–
Done	–	–	–	–	–	x	–
Failed	–	–	–	–	–	x	x

Table 2. Predecessors' state of `TEST` task

	predecessors' state						
Test	InD	Wai	Pln	Act	Spd	Dne	Fai
InDefinition	x	x	x	x	x	x	x
Waiting	x	x	x	x	x	x	–
Planning	–	–	x	x	x	x	–
Active	–	–	x	x	x	x	–
Suspended	–	–	x	x	x	x	–
Done	–	–	–	–	–	x	–
Failed	–	–	–	–	–	x	x

Table 3. Trigger definition sample

Event	`DESIGN`	`TEST`
`Pred.Failed`	`suspend;` `suspend successors`	`notify actor`
`TokenRelease`	`Consume(Token)`	`notify actor`

`RevokeRel.`	`UnConsume(Token)`
	`UnRelease(AllToken)`
	`Suspend(self)`
		`UnConsume(Token)`
		`notify actor`

...	...	...

When technical developers or the project manager manipulate the actual task net, each manipulation raises an *event*. Examples of such events are listed in the first column of table 3. Such actions affect the successors and subprocesses of the task. Appropriate reactions for a task which receives an event can only be defined when the specific scenario is known. Too many factors may influence a suitable reaction, so the process modeler may specify for each type individual triggers. For the `DESIGN` and the `TEST` task type, trigger definitions are presented. In case of receiving the `PredecessorFailed` event, `DESIGN` reacts with suspension of itself and subsequently all its successors. Suspended tasks may be resumed as soon as the consequences become clear. If a test receives the `PredecessorFailed` event, only its actor is notified. The second example is the `TokenRelease` event. If a design task receives such an event, a new document

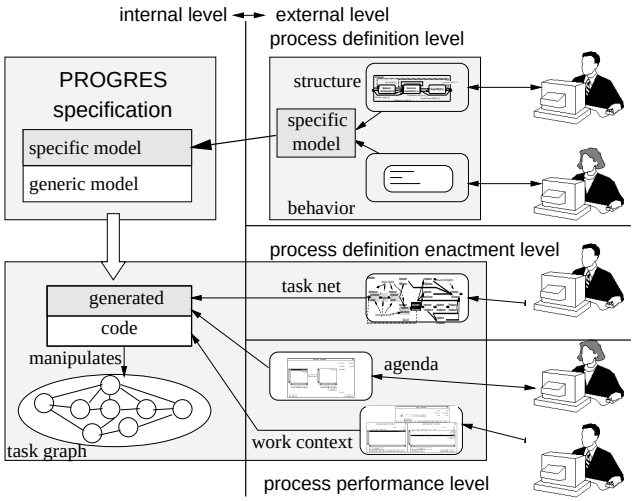

Figure 6. Realization overview

version is consumed in order to obtain the new requirements quickly. For the `TEST` task, only the actor is notified.

A user defined adaption of the process management system as it was described above is sometimes hardly possible. In a project which is completely new for a company, the knowledge required for a successful adaption would be missing. Thus, process modelers are allowed to specify as much or as little information as desired. In the extreme case, it is possible to build task nets without any adaption. For this reason, standard types are introduced. Instances of such types (tasks, data flows, parameters, etc.) underly no restriction, i.e. there are no cardinality constraints, state conditions, etc.

6. Realization Approach

After we presented the tool environment for the different user roles, we sketch the realization approach of the DYNAMITE model briefly. Figure 6 provides an overview. The vertical line distinguishes between the external and the internal level. The internal level is concerned with the complex data structures maintained by the management system. The process management tools are on the external level and are user friendly views onto these data structures.

On the internal side of the process definition level, a generic model is provided, which is defined by a PROGRES specification. It is independent of a specific application domain and defines the constituents of task nets, general structural constraints, and basic operations for editing, analyzing, and executing. The specific adaptations from the external side are transformed into PROGRES specification code, as well. The code pieces augment the generic model with the application-specific structural and behavioral knowledge.

A cutout of such a PROGRES specification is on a high level of abstraction, which was presented in [10]. Instead of describing a graph transformation in terms of elementary operations such as creation and deletion of single nodes and edges, a graph rewrite rule describes replacement of a whole subgraph declaratively. The PROGRES compiler takes care of all algorithmic details for graph transformations effectively [22]. We emphasize the use of the formal specification language, which is well suited to develop prototypes. Design decisions on this level of modeling can be easily revoked and are much easier to handle than in a programming language or in a database system.

The double arrow on the left side symbolizes the generation of source code from the formal specification. The generated code is linked into a framework for manipulating persistent graph-based data structures. The prototype obtained offers an interface on which the different tools for developers and process managers are installed.

The tool for the project manager is based on the source code, as well. With the help of a standard frame architecture for interactive systems, the management environment can be compiled. It offers a graphical user interface based on the toolkit Tcl/Tk [17]. Different views are defined in the specification which present only relevant information from the complex graph transformation. Pop-up menus are offered for inserting new tasks, relations, etc.

Portability and model independence have been important aims when designing the tools for developers. To make the implementation independent both from changes in the task graph implementation and the process model parameterization, a high level interface onto the task graph has been defined. This interface offers queries and modification operations. Via query results, not only the contents of task and document lists, but also menu entries and selection lists are determined. In this way, the adaptation as in table 2 controls which task state transitions are offered. External tools are called in a uniform way. A tool specific *wrapper* sets up a temporary session workspace and takes care of tool parameters, filename conventions, etc.

7. Conclusion

We have sketched an environment for managing software processes that is highly adaptable, both before and during execution. External views for different user groups are customized according to their infor-

mation needs; internally, graph grammar specifications are used.

Model and tool development for managing software processes are part of other research activities of our department. These include not only management of development activities [16], but also building interactive, graph-based tools from specifications. To the system described here, other groups in our department have contributed the PROGRES environment, the GRAS database management system [15], and numerous subsystems for building graph-based tools. We are constantly working with them in order to make these systems better suited to our needs.

The generic process model has been completely specified in PROGRES. The specification consists of 130 rewrite rules and 100 node classes. It has reached a size of 80 pages. Besides the generic model, we have specified adaptable models for products (documents, versions, configurations and their relationships) [20] and resources (resource configurations, actual resources, required resources). Future work will address integration of the three models and the construction of an integrated tool set.

The management tool which is currently under construction is based on the prototype generated from the overall specification (generic and specific part). The graphical presentation, especially the automatic layout, will be made easier to understand. Furthermore, a useful environment has to offer several levels of detail and allow to zoom in areas of interest. Integration of tools for task, product, and resource management is of special importance here. Analysis tools must help the manager to pinpoint and correct problems, like resource conflicts, approaching deadlines etc. easily.

The tool for developers has been realized on a variety of Unix systems. Just like in other areas of the system, we intend to use general modules with generator-produced extensions to connect the frontend to the task graph. These modules will replace the currently hand-written abstraction modules. The product model, a manager's tool set, and the developer's frontend described above have been applied to development in the mechanical engineering domain.

So far, we have parameterized the process model on the level of detail of the PROGRES language and then used the PROGRES tools to generate a prototype of the management system. While this is the right level for an experienced modeler and indispensable for sophisticated adaptations, we intend to define and implement more user-friendly tools for process modelers. These tools will work on a higher level of abstraction. They will allow to select from predefined behaviors for every part of the model and thus reduce the training time. Although we have presented some ideas of an user defined adaptation, we are still searching for a suitable representation.

References

[1] S. Bandinelli, A. Fuggetta, C. Ghezzi, and L. Lavazza. SPADE: An Environment for Software Process Analysis, Design, and Enactment. In Finkelstein et al. [8], chapter 9, pages 223–247.

[2] G. Canals, N. Boudjlida, J.-C. Derniame, C. Godart, and J. Lonchamp. ALF: A Framework for Building Process-Centered Software Engineering Environments. In Finkelstein et al. [8], pages 153–185.

[3] R. Conradi, M. Hagaseth, J.-O. Larsen, M. N. Nguyên, B. P. Munch, P. H. Westby, W. Zhu, M. L. Jaccheri, and C. Liu. EPOS: Object-Oriented Cooperative Process Modelling. In Finkelstein et al. [8], chapter 3, pages 33–70.

[4] B. Curtis, M. Kellner, and J. Over. Process Modeling. *Communications of the ACM*, 35(9):75–90, 1992.

[5] W. Deiters and V. Gruhn. Managing Software Processes in MELMAC. *ACM Software Engineering Notes*, 19(6):193–205, 1990.

[6] M. Dowson and C. Fernström. Towards requirements for enactment mechanisms. In B. C. Warboys, editor, *Proceedings of the Third European Workshop on Software Process Technology*, pages 90–106. Lecture Notes in Computer Science Nr. 772, Springer–Verlag, Feb. 1994.

[7] C. Fernström. PROCESS WEAVER: Adding Process Support to UNIX. In *2^{nd} International Conference on the Software Process: Continuous Software Process Improvement*, pages 12–26, Berlin, Germany, Feb. 1993. IEEE Computer Society Press.

[8] A. Finkelstein, J. Kramer, and B. Nuseibeh, editors. *Software Process Modelling and Technology*. Research Studies Press, Taunton, Somerset, England, 1994.

[9] M. A. Gisi and G. E. Kaiser. Extending a tool integration language. Technical Report CUCS-014-91, University of Columbia, 1991.

[10] P. Heimann, G. Joeris, C.-A. Krapp, and B. Westfechtel. A programmed graph rewriting system for software process management. In *Proceedings Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation (SEGRAGRA '95)*, volume 2 of *Electronic Notes in Theoretical Computer Science*, 1995.

[11] P. Heimann, G. Joeris, C.-A. Krapp, and B. Westfechtel. DYNAMITE: Dynamic Task Nets for Software Process Management. In *Proceedings of the 18^{th} International Conference on Software Engineering*, pages 331–341, Berlin, Germany, 1996. IEEE Computer Society Press.

[12] G. T. Heineman, G. E. Kaiser, N. S. Barghouti, and I. Z. Ben-Shaul. Rule Chaining in MARVEL: Dynamic Binding of Parameters. *IEEE Expert*, 7(6):26–32, Dec. 1992.

[13] G. Junkermann, B. Peuschel, W. Schäfer, and S. Wolf. MERLIN: Supporting Cooperation in Software Development Through a Knowledge-Based Environment. In Finkelstein et al. [8], chapter 5, pages 103–129.

[14] M. I. Kellner. Software process modeling support for management planning and control. In M. Dowson, editor, *Proceedings of the First International Conference on the Software Process*, pages 8–28. IEEE Computer Society Press, Aug. 1991.

[15] N. Kiesel, A. Schürr, and B. Westfechtel. GRAS: A Graph-Oriented (Software) Engineering Database System. *Information Systems*, 20(1):21–52, 1995.

[16] M. Nagl and B. Westfechtel. A Universal Component for the Administration in Distributed and Integrated Development Environments. Technical Report 94-08, RWTH Aachen, D-52056 Aachen, 1994.

[17] J. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley Pub., 1994.

[18] A. Schürr. Rapid Programming with Graph Rewrite Rules. In *USENIX Symposium on Very High Level Languages*, pages 83–100. USENIX Association, 1994.

[19] A. Schürr, A. J. Winter, and A. Zündorf. Graph Grammar Engineering with PROGRES. In W. Schäfer and P. Botella, editors, *Proc. of the* 5^{th} *European Software Engineering Conference (ESEC)*, volume 989 of *Lecture Notes in Computer Science*, pages 219–234, Sitges, Spain, 1995. Springer-Verlag.

[20] B. Westfechtel. A Graph-Based System for Managing Configurations of Engineering Design Documents. *International Journal of Software Engineering & Knowledge Engineering*, 6(4):549–583, 1996.

[21] P. Young. *Customizable Process Specification and Enactment for Technical and Non-Technical Users.* PhD thesis, University of California Irvine, 1994.

[22] A. Zündorf. Graph Pattern Matching in PROGRES. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Proceedings of the* 5^{th} *Intern. Workshop on Graph Grammars and Their Application to Computer Science*, volume 1073, pages 454–468, 1996.