Integrating Planning and Execution in Software Development Processes*

Barbara Dellen & Frank Maurer
AG Expert Systems, University of Kaiserslautern, Postfach 3049, 67653 Kaiserslautern, Germany
e-Mail: {dellen, maurer}@informatik.uni-kl.de

Abstract

In conventional approaches to support planning of software development processes, the project plan has to be completely specified ahead of the process enactment. These approaches ignore two main characteristics of software development processes: Firstly, many planning decisions can only be made on base of knowledge resulting from the development process itself. Therefore the project planner must be able to extend and adapt the plan during enactment on basis of the current project data. Secondly, plan changes have to be taken into account. Considering these observations, the planner can be supported effectively, only if planning and enactment steps are alternated.

In this contribution we identify the main requirements on a "dynamic" planning of software development processes and present methods and techniques which meet them.

1 Introduction

Software development processes are in general characterized by a high complexity and a large number of people involved. These characteristics lead to some problems. To handle the complexity, the project has to be decomposed into partially interdependent processes. The processes have to be distributed among the employees involved in the project. Coordination of the activities of the employees is entailed with high effort. The complex causal dependencies between processes have to be considered when project changes occur.

A way out of these problems is an accompanying planning of the development process. Identifying processes, formulating their goals, assigning resources to them, and the definition of causal dependencies between processes and data results in a better understanding of the project. As a result, project coordination effort is reduced and project costs can be better estimated.

The idea of a computer based project support is to reduce the work load of the project members by bookkeeping the project enactment, and guiding the involved people. Some systems for planning and enactment support of software processes have been developed [10]. The weakness of these approaches is, that alternating of planning and enactment steps is not sufficiently supported. Unfortunately greater flexibility of project planning and enactment is essential for the following reasons:

- A detailed planning needs knowledge resulting from the development process itself. Therefore the plan has to be completed during enactment using the project specific knowledge. In Software Engineering processes, for example, the planning of the design can only start after the requirements are extracted from the problem description of the customer. Therefore, the development environment has to allow the extension of the initial plan by a detailed planning during enactment.
- Changing conditions or planning errors force to replan parts of the project no matter how far the enactment is advanced. For example, most plans built up are not error free. Missing data and inappropriate solution alternatives available during enactment point to planning errors. It must be possible to correct them if the plan is already in enaction.

Our research group is developing methods and techniques for computer based support of distributed planning and enactment of design processes [16, 17, 18]. One goal

*This work is supported by the *Deutsche Forschungsgemeinschaft, Sonderforschungsbereich 501*

1080-1383/96 $5.00 © 1996 IEEE
Proceedings of WET ICE '96

of our research activities and the focus of this contribution is to alternate project planning and enactment.

The contribution is organized as follows: First we present concepts and techniques to describe project plans. After that we deal with the question how to interpret and enact these plans. In chapter 3 we summarize how our fundamental concepts fit the requirements on an alternated planning and enactment of software development processes.

2 Planning and enactment of software development processes

In our approach project plans are represented by explicit process and product models which can be created within a *Modeler.* The plans are enacted by a workflow engine, called *Scheduler.* The Scheduler interprets the models and manages the process information, necessary to give useful planning and execution support. An inherent feature of our approach is the management of causal dependencies, which are necessary to support project changes, project coordination, and tracing of processes. The dependencies are automatically extracted from the process models and managed by the Scheduler. In this contribution we only give a brief overview over the dependency management mechanism as far as necessary to understand our concepts to alternate planning and enactment.

The next two sections shortly describe the basic language concepts the Modeler provides to generate project plans and give a survey of the functionality of the Scheduler. We clarify what kind of information is managed and how it is done.

2.1 Modeling software processes

For modeling project plans we provide a set of language concepts. The project plan essentially corresponds to the *Model of Expertise,* and the *Agent Model* of *CommonKADS* [3, 11], known from Knowledge Engineering, extended by our needs.

The basic language concepts are processes, methods, products and resources. A short description of them is given now, for details see [15].

Processes (Tasks). A process defines a *goal to be reached.* Every process uses a set of input products and produces a set of output products. A product may be any kind of data. In the plan, the products are referenced by typed formal parameters. Furthermore a set of methods and a list of skills, an agent who wants to work on the task must fulfill, is assigned to every process.

Methods. Every method describes a *way how to solve a process.* We distinguish between *atomic* and *complex* methods. A complex method decomposes a process into one or more subprocesses and specifies the product flow between them. The direction of the product flow is specified by the use/produce relations of the subprocesses: to define a product flow dependency between two processes, an output product of one process is related to an input product of the other one.
Atomic methods represent the creation of products.

Products (Concepts). To represent products we use an object oriented approach. We distinguish between product *classes* and product *instances.* Product classes define the type of formal parameters. Formal parameters are container for the product instances, produced during the enactment. The product instances are the data itself. Product instances are created during execution by applying atomic methods. The type of a product instance is given by the product class it belongs to.

Resources (Agents). We distinguish between two types of agents: *machines* and *human* agents. Every agent is characterized by a set of skills. During enactment, every process is delegated to a set of potential agents but at last solved by exactly one.

2.2 Enacting software processes

The process models are carried out with support of the Scheduler. The Scheduler is a bookkeeping unit, that stores and manages relevant project information. The agents can access the information they need. Changes in the project state are actively propagated to them. Besides the Scheduler filters the propagated messages: only relevant information reaches the users, and only affected agents are informed. At a close look the Scheduler

- manages the current state of processes, methods and products,
- manages decisions concerning the method selection or rejection,
- stores all products produced during process enactment,
- manages causal dependencies between processes, methods and products,
- notifies affected team members about plan changes,
- manages "to do" agendas for the team members,
- discovers and solves inconsistencies in the enactment process.

Starting the execution the Scheduler step by step automatically deduces the required process information (such as processes, methods, and resource assignments) and causal dependencies from the process model, and manages them. The deduced structures are the basis of a dynamic planning and allow to coordinate the activities of the agents.

The management of causal dependencies which describe how changes influence the current project is

essential to an integrated planning and enactment support. To propagate the consequences of changes (to the affected agents), the dependencies have to be managed with a suited mechanism [19].

The next section gives insight in the kind of causal dependencies handled by our system and focuses on formalisms underlying the dependency management. In this contribution we focus on those parts of our approach, which are necessary to allow the integration of planning and enactment.

2.3 Deducing and managing causal dependencies

To achieve traceability and to support change processes, the explicit representation and management of causal dependencies is a prerequisite. In our approach dependencies between *decisions, processes, assignments, and delegations* [8, 17] are represented.

An *assignment* assigns a concrete product (or *product instance*) to a formal parameter.

If a method is applied, a *decision* is made. To solve a process, the agent decides for one of the applicable methods. A team member decides for example to make an object oriented design that results in the decomposition of the process *Object Oriented Design* in subprocesses *Define Class Hierarchy*, *Define Messages*, and *Define Inheritance Relations*.

The decision for an atomic method results in the instantiation of a set of product classes and their assignment to related parameters.

There is a 1:1 relation between a decision and a method. If a method contributes to the actual project plan, the referencing decision is valid. If decisions are mistaken or unsuited they can be *rejected*. The rejection of a decision for a method m_i is described by a term *rejected decision*(m_i) and means that there is a reason against this decision. A decision and its rejection are semantically interdependent: if there are rationales against the decision it is rejected (for details see [8, 7]). Specifying this semantic allows us

- to determine, if a method contributes to the actual project plan,
- to handle conditions for the validity/invalidity of a decision, and
- to state rationales for and against the validity of processes and products.

Altogether, we define the following semantic for the validity of a decision: A decision is valid if all rationales pro a decision are valid, and no rationale for its rejection is valid. If you are interested in the different types of rationales which can be formulated in our approach, see [7].

Processes, decisions, rejection of decisions and assignments are causal interdependent. The underlying semantic of the project plans allows us to explicit deduce and manage causal dependencies between them.

We distinguish between four kind of dependencies:

Dependencies between decisions and product assignments. As described, the decision for an atomic method assigns products to formal parameters. If the decision for an atomic method is rejected for any reason, the related products are invalidated. The corresponding formal parameter becomes unassigned again. This behavior is used to manage product flow dependencies (see below).

Dependencies between processes & subprocesses. Complex methods decompose processes into a set of subprocesses. If an agent decides to apply a complex method, the set of subprocesses related to the method becomes part of the execution process and has to be solved. If this decision is rejected later, the subprocesses become invalid. Decisions, which have been taken within the subprocesses, must be rejected, too. Those agents who are working on invalidated processes are automatically informed.

Dependencies deduced from the product flow. Decisions are made within the context of the actual inputs of a process. Therefore the decision for a method is bound to the current assignments of the input parameters. In general the input products have been produced by applying atomic methods within a preceding processes in the development process. As described, product instances can be removed from the repository by retracting the corresponding decision. As a result, those decisions, which base on the context of the now invalid products have to be retracted, too. This behavior can spread through a chain of decisions with partly global effects on the development process. Affected agents are informed of such changes. With the project knowledge provided by the Scheduler, they can react on the changes appropriately.

Delegation dependencies. During execution, every process is delegated to a set of agents. This set specifies those agents, which are allowed to work on the process. A process has been delegated to an agent, if three conditions hold: The agent is qualified, the project manager has selected him, and he is available. The first condition is automatically checked by the Scheduler. It compares the skills required to work on the process with the abilities of the agent. The second conditions holds, if the project planner has assigned the process to an agent. The last condition holds, if the agent for example is not on vacation. This condition can be checked automatically, if the Scheduler has access to vacation tables or something similar. Because of changing conditions, e.g. changed process descriptions in the model or in case of illness, a delegation may become invalid. By managing delegation dependencies, the project manager can be informed of such changes.

The described dependencies can be formulated as logical implications (see [8, 7]). By the explicit representation of the dependencies, the scheduler is able to provide the agents with the actual process data for decision, and planning support [8, 19], such as

- products produced within preceding enactment steps,
- knowledge about (rejected) alternatives and their rationales,
- knowledge about current delegations and changed delegations, and
- knowledge about the current state of processes.

3 Alternating planning and execution

Managing causal dependencies is essential to alternate planning and enactment. On one hand, effects of changes can be propagated to dependent processes and decisions. Affected project members are notified. On the other hand, relevant project information is provided to the project members (i.e. planners, developers) at the right place and the right time. Therefore, concrete project knowledge can be used to plan further project steps.

Our approach allows to alternate planning and enactment steps in three ways. Firstly, the process model can be refined, and modified during enactment. Secondly, the initial project plan is specialized during enactment by applying methods and delegating processes. Thirdly, the plan can be adapted to the current situation by changing (i.e. making and rejecting) decisions.

3.1 Making and changing decisions

There are two characteristics of the Modeler, that increase the flexibility of planning:

(1) methods can be defined, but do not have to be applied at the same moment.

(2) processes need not be delegated to concrete agents. Only required skills are assigned to processes.

By delaying the selection of methods and the delegation, concrete project knowledge can be used for decision support. As a result, decisions are dependent on the current situation. If the context of a decision changes, the decision can be rejected by the agent, and replaced by another decision. Our approach allows an agent to

- *make decisions.* Working on a process, the agent can choose one of the applicable methods specified in the model. Methods rejected in the actual context are not available to the agents. As already mentioned, the responsible agent decides during execution for one of the alternatives. An agent is, for example, responsible for testing a developed program. Input for the process is the developed program. The model provides two testing alternatives, *equivalence class based testing* and *code reading*. Because the program is well written, the agent decides for code reading. The actual input supports him to find a situation related solution.
- *reject decisions.* During enactment an agent has the possibility to reject valid decisions. Because of the established product flow dependencies and process dependencies such a replanning step can have consequences for the work of other agents. Dependent decisions, subprocesses and products become invalid. If, for example, a developer designs a *software component*, and a part of the component requirements is removed, the component has to be redesigned. Because the Scheduler manages product flow dependencies, the affected agents are informed and can react on the changes.
- *delegate processes.* As described in section 2.3, the project manager can assign agents to processes. He can only select those agents, who are qualified and available. The manager returns the result of the delegation to the Scheduler, which adds the delegated process to the "to do" agendas of the selected agents.
- *reject delegations.* Rejecting a delegation may be necessary if time and personnel conditions change (for example an employee leaves the enterprise during project enactment). For this, one can reject the delegation of an agent to a process.

3.2 Refining and modifying project plans

In some situation it is insufficient to solve a problem by selecting or changing given methods, but the underlying model has to be refined or modified. In our approach, the process model can be refined, and modified any time in the enactment: new methods can be added to a process, not yet completely specified methods can be modeled on a fine granularity, the product flow of a method can be modified, and modeling errors can be corrected. The Scheduler reinterprets the model modifications and makes them available to the current project. An agent, working on a process, has the following possibilities to change the model:

- *Adding a new solution alternative.* During enactment, increasing process knowledge may result in new solution approaches. The agent easily can establish new methods and add them to the model. He has to define the product flow for the method and assign the method to an appropriate process. Because the Scheduler bases on interpretative techniques, the new method is immediately available to the current project.
- *Refining the model.* Many methods can be planned only on basis of project knowledge. Those methods stay unspecified in the initial model. The *architectural design* of a software product, for example, cannot be planned before the *requirements document* has been

produced. If the required products are produced by preceding activities the scheduler recognizes it and informs the responsible agents. The project planner extends the plan by adding new processes and parameters, specifying the product flow between them, and refining the subprocesses within the method definition using the available product data. The changes are automatically interpreted and included into the Scheduler.

- *Correcting faulty models.* Many modeling errors are noticed not before the plan is already in execution. Nevertheless the errors have to be corrected to guarantee a correct execution. For example, the product flow between the processes *architectural design* and *implementation* may have the wrong direction: the process *architectural design* consumes wrongly the already implemented program. To guarantee a correct enactment, the project planner is forced to correct the product flow within the model.
- *Changing agent bindings.* Skills of agents may change, as well as skills that are necessary to work on a process. In both cases, the project manager can adapt the corresponding parts of the model.

4 Example

The example is taken from the Software Engineering domain and gives an impression how CoMo-Kit supports the method selection and model extension.

Figure 1 shows a small part of a model "IEEE Standard for Developing Software Life Cycle Processes" [5] developed by IEEE and modeled and extended within the SFB 501[1].

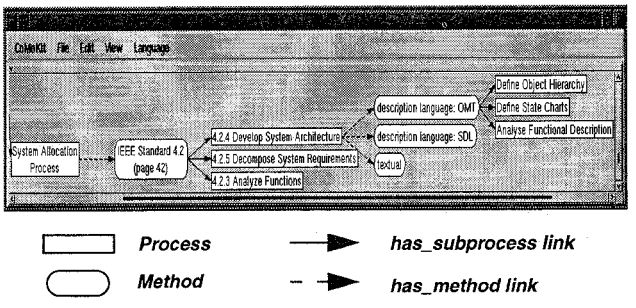

Fig. 1: Part of the *IEEE* process and method hierarchy

The abstract process *System Allocation Process* is solved by method *IEEE Standard 4.2*, which decomposes the *System Allocation Process* into the subprocesses *4.2.5 Decompose System Requirements*, *4.2.3 Analyse Functions* and *4.2.4 Develop System Architecture*. The product flow defined on this method is shown in figure 2. One can see, that task *4.2.4 Develop System Architecture* uses the document *Functional Description of the System,* produced by process *4.2.3 Analyse Functions.*

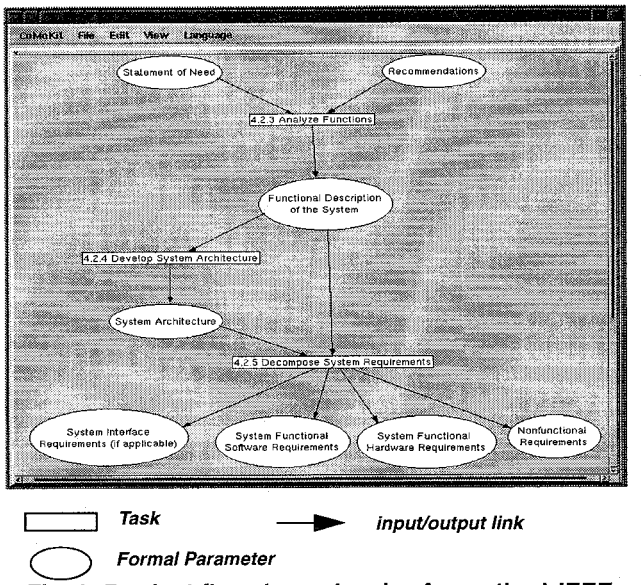

Fig. 2: Product flow dependencies for method *IEEE Standard 4.2*

During enactment the current assignment of this input can be used to decide for one of the three available methods to solve process *4.2.4 Develop System Architecture*, namely *description language: OMT, description language: SDL, textual* (see figure 1) or to specify a new method. The window that displays this information to the human agent is shown in figure 3. The left side of the window displays the applicable alternatives to solve task *4.2.4 Develop System Architecture.* It provides some buttons which allow to define a new or to change an existing method. Furthermore the agent can apply a method by pushing the button "select". On the right side of the window, the input parameters and their actual assignments are displayed. In the example, the input *Functional Description of the System* is a FrameMaker document with name "functional_description.doc" that can be opened by pushing button "open".

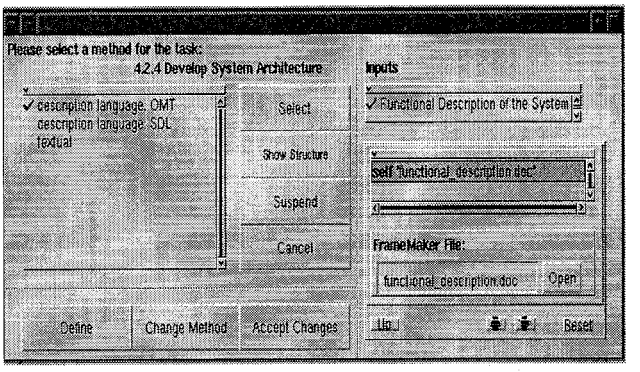

Fig. 3: View on the method selection window of an agent

1. Sonderforschungsbereich 501 „Development of large systems with generic methods"

In figure 1 one can see, that method *textual* defines no subprocesses yet. The planner delayed the specification of this method to the execution. With the concrete input document "functional_description.doc", the method can be completed now. In the situation shown in figure 3, the agent might complete the modeling of method *textual* on basis of the input product "functional_description.doc". The agent has to define subprocesses and the product flow between them, considering the inputs and outputs of the superprocess *4.2.4 Develop System Architecture* (see figure 2). A possible decomposition of the method is shown in figure 4.

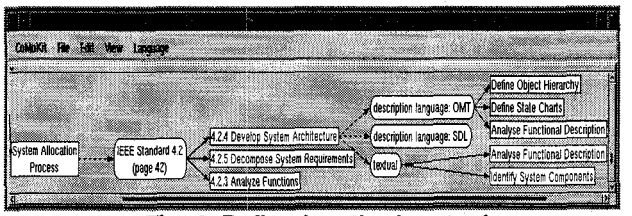

Fig. 4: Refined method *textual*

The product flow of method textual is shown in figure 5. After the agent accepted the changes on the model by pushing button "accept changes" (figure 3), they are tied into the dependency management mechanism of the Scheduler. From now on, the selection of method *textual* leads to the decomposition of process *4.2.4 Develop System Architecture* into subprocesses *Analyse Functional Description* and *Identify System Components* with respect to the specified product flow.

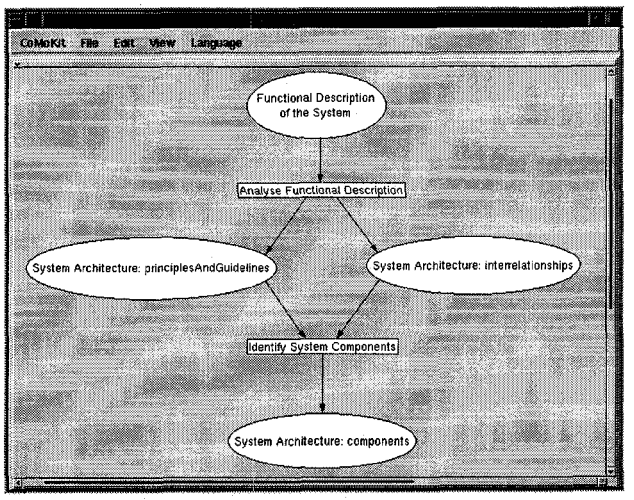

Fig. 5: Product flow of method *textual*

5 Related work

In software engineering research several approaches which support modeling and the enactment of software engineering processes exist. Two of them, Marvel [13] and GRAPPLE [12], integrate AI techniques similar to our CoMo-Kit approach.

Marvel follows a rule-based approach to express assumptions for the enactment of process steps. The approach supports forward-chaining of rules as well as backward reasoning. The latter tests which actions must be undertaken to fulfil the precondition of the current action. Marvel does not support justifications for choosing a method nor is it possible to express causal dependencies between products and decisions.

GRAPPLE is based on an explicit model of planning. Using an operator-based language supports the representation of preconditions of actions. GRAPPLE plans in a goal-oriented manner. It contains a plan recognition component which interprets user actions and integrates them into the actual plan. In GRAPPLE, it is assumed that, for planning and plan recognition, knowledge is needed which is not included in operator definitions. This knowledge is given as a set of assumptions and handled by a Reason Maintenance System. Contrary to our approach, this knowledge is mainly used to constrain the set of applicable operators. Causal Dependencies between products are not explicitly represented in GRAPPLE. GRAPPLE does not allow for alternating planning and enactment.

The language MVP-L [4] supports the modeling and enactment of software processes. The MVP-System does not handle causal dependencies and therefore change propagation is not supported. Currently, we integrate CoMo-Kit with the MVP approach [22].

The SPADE environment is a system for developing analyzing, and enacting process models described in SLANG [2]. *Activities* are modules with well-defined interfaces and a Petri net specification as a body. Activity types may be changed during enaction but they do not affect existing instances. When the type of an active process is modified, SPADE prompts the user to provide a transformation function. The user must decide when to start process evolution. The system does not provide any support to decide which parts need to be changed.

The database-oriented EPOS Process Modeling System distinguishes between classes (templates), instances thereof, and information about the creation, change, and conversion of classes and instances on a meta-level [14]. Feedback about correctness and performance of the enacted process model triggers changes of classes and instances which are under version control. Classes and instances may be changed in the case of inactive processes. The user is responsible for establishing consistency between classes and instances. No dependencies between process fragments are managed so it is not possible to determine what processes accessed a faulty product and might be enacted another time. Detection of deviations

and recognition of a change's impact are completely left to the user.

Redoing [21] is an operation in the Hierarchical and Functional Software Process (HFSP) approach that allows cancellation of erroneous activities and doing that part of the process again. Software development processes are understood as functions organized in a hierarchy (called an enaction tree). Redoing means cutting a subtree out of the enaction tree and replacing it with another tree which is newly enacted. The decision to redo is specified in the process models. The process models must be completely defined before interpretation. The decisions for redoing are predefined, which means that criteria to detect deviations from the plan must be specified within the models.

6 Conclusion & current work

Within the CoMo-Kit system we have developed methods and techniques which integrate planning and enactment of software processes. The initial plans can be specialized, refined and adapted during process execution.

We are evaluating our research results in urban land planning [18] and Software Engineering domains within the Sonderforschungsbereich 501, "Development of large systems with generic methods" project B2, "Knowledge based planning and control of software development processes.

The CoMo-Kit Modeler is fully implemented as well as the CoMo-Kit Scheduler for the local area network. We just connected CoMo-Kit to the world wide web to support globally distributed software development processes.

An open problem is how to react on model modifications that affect parts of the plan that are already executed. Solving this problem is the current goal of our work.

References

[1] J. Angele, D. Fensel, D. Landes, S. Neubert, R. Studer: Model-based and Incremental Knowledge Engineering: The MIKE Approach. In *Knowledge Oriented Software Design*, J. Cuena, ed. IFIP Transactions A-27, Elsevier, Amsterdam, 1993, 139-168.

[2] Sergio C. Bandinelli, Alfonso Fuggetta, and Carlo Ghezzi. Software process model evolution in the SPADE environment. *IEEE Transactions on Software Engineering*, 19(12):1128–1144, December 1993.

[3] J. Breuker, W. van de Velde (eds.): CommonKADS Library for Expertise Modelling, IOS Press, 1994.

[4] A. Bröckers, C. M. Lott, H. D. Rombach, and M. Verlage. MVP–L language report version 2. Technical Report 265/95, Department of Computer Science, University of Kaiserslautern, 67653 Kaiserslautern, Germany, 1995.

[5] IEEE Standard for Developing Software Life Cycle Processes (IEEE Std 1074-1991). *Institute of Electrical and Electronics Engineers, Inc.*,New York/USA, 1992.

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

[7] B. Dellen, K. Kohler, F. Maurer: Integrating Software Process Models and Design Rationales. Will be published on *Proceedings of KBSE 96*.

[8] B. Dellen, F. Maurer, J. Paulokat: Verwaltung von Abhängigkeiten in kooperativen, wissensbasierten Arbeitsabläufen. In *Proceedings of the 3. deutschen Expertensystemtagung*, pages 72-89, 1995

[9] Doyle, J.: A Truth Maintenance System, *Artificial Intelligence*, 12:231-272, 1979.

[10] A. Fuggetta: A Classification of CASE Technology. *Computer*, Vol. 26, No 12, December 1993

[11] R. de Hoog, W. Menezes, C. Toussaint, B. J. Wielinga, R. M. Taylor, C. Bright, W. van de Velde: The CommonKADS model set, ESPRIT Project P 5248 KADS-II/M1/DM..1b/UvA/018/5.0, University of Amsterdam, Lloyd's Register, Touche Ross Management Consultants & Free University of Brussels.

[12] Karen Erikson Huff: Plan-Based Intelligent Assistance: An Approach to Support the Development Process. PhD thesis, University of Massachusetts, September 1989

[13] G.E. Kaiser P.H. Feiler, S.S. Popovich: Intelligent Assistance for Software Development and Maintenance, *IEEE Software*, May 1988.

[14] M. Letizia Jaccheri and Reidar Conradi. Techniques for process model evolution in EPOS. *IEEE Transactions on Software Engineering*, 19(12):1145–1156, December 1993.

[15] Maurer, F.: Project Coordination in Design Processes, WET ICE 96, Workshop "Project Coordination",

[16] Maurer, F.: Hypermedia-Based Knowledge Engineering for Distributed Knowledge-Based Systems (in german), PhD Universität Kaiserslautern, 1993, auch: DISKI 48, infix-Verlag, ISBN 3-929037-48-3.

[17] Maurer, F., Paulokat, J.: Operationalizing Conceptual Models Based on a Model of Dependencies, in: A. Cohn (Ed.): ECAI 94. 11th European Conference on Artificial Intelligence, 1994, John Wiley & Sons, Ltd.

[18] Maurer, F., Pews, G.: Ein Knowledge-Engineering-Ansatz für kooperatives Design am Beispiel der Bebauungsplanung, Themenheft Knowledge Engineering, *KI 1/95*, interdata Verlag, 1995.

[19] Petrie, Ch.: Planning and Replanning with Reason Maintenance, PhD, University of Texas, Austin, 1991.

[20] F. Maurer: Modeling the Knowledge Engineering Process, *2nd Knowledge Engineering Forum 96*.

[21] Masato Suzuki, Atsushi Iwai, and Takuya Katayama. A formal model of re–execution in software process. In Leon J. Osterweil, editor, *Proceedings of the 2nd International Conference on the Software Process*, pages 84–99. IEEE, IEEE CS Press, February 1993.

[22] M. Verlage, B. Dellen, F. Maurer, J. Münch: A Synthesis of two Process Support Approaches, *Proceedings of SEKE-96*.