

Project Coordination in Design Processes⁺

Frank Maurer
University of Kaiserslautern
P.O. Box 3049
67653 Kaiserslautern
e-mail maurer@informatik.uni-kl.de

Abstract

In this paper we describe how explicit models of design processes can be used to guide and control distributed development of complex systems. The paper focuses on

- *techniques which automatically infer dependencies between decisions from a process model and*
- *methods which allow to integrate planning and execution steps.*

Managing dependencies between decisions is a basis for improving the traceability of development processes. Switching between planning and execution of subprocesses is an inherent feature in the development of large complex systems.

The paper concludes with a description of the CoMo-Kit system which implements the technologies mentioned above and which will use WWW technology to coordinate development processes.

1 Introduction

The CoMo-Kit project of the University of Kaiserslautern focuses on methods and techniques which support project planning and project coordination for complex, distributed development and/or design projects. To evaluate our approach we work on two application domains: software engineering, which is the running example here and also in [10], and urban land-use planning (cf. [20][22]).

In the future, the development of large systems will be a typical task for virtual enterprises: People with different educational background (e.g. economics, computer science, arts for interface design) working on many locations all over the world (including emerging market countries like India or Eastern Europe) will have to work together to fulfill the business needs.

A deeper look on these applications shows that large scale design processes in virtual enterprises can be characterized by the following features:

- People work on different locations and at different times on a common task. Therefore, time and money are spent for planning and coordination of the work process.
- Large-scale development processes are long-term processes. Requirements on the outcome may and will change during process execution. Change processes are a central reason for running out of project schedule and budget.
- During the process, a lot of decisions are taken by each participant. These decisions influence each other and base on each other.

These features lead to certain requirements for process-supporting tools:

- The tool has to support planning, coordination, and execution of the work process.
- Spatial and chronological separation of the tasks necessitates that the tool serves as a kind of "intelligent memory" of its users, in order to allow participants to understand the rationales behind decisions.
- If changes take place, the tool should automatically inform the involved users.

The next chapter gives an overview on the architecture of the CoMo-Kit System which implements our concepts.

These resulted from an abstract view on the questions which must be supported by a work process coordination tool:

- Who should do what?

This question addresses the problem of project planning and the delegation of tasks to agents. A framework for project planning is described in chapter 3.

- When can the work on a task start?

Here, we distinguish between conditions which must be

⁺This work was sponsored in part by the Deutsche Forschungsgemeinschaft as part of the special research activity Sonderforschungsbereich 501 „Development of large systems with generic methods“

fulfilled for planning the work on a task and for execution of the task itself. Chapter 4 gives further details.

- Who must be informed about changes?

To improve the traceability of decisions is the core of our answer (see chapter 5).

The state of the implementation is described in chapter 6. The last chapter gives a short summary and describes our current and future work.

2 CoMo-Kit System Architecture

Within the CoMo-Kit project of the University of Kaiserslautern techniques, methods, and systems are developed which support planing and enacting complex distributed cooperative design processes [21], [20]. For a new development project, in a first step an *initial project plan* is created. This plan contains descriptions of the tasks to be done, a definition of the data structures which must be created during task execution and a list of the team members involved in the design process. *Our approach allows to interlock planning and enactment.* This is a main feature for design processes because they only can be planned on a detailed level when some steps have already been carried out. This is one reason why normal workflow systems cannot be used for design tasks: They require that the task flow is defined *before* the enactment starts.

The current project plan is used by our Workflow Management Server, the CoMo-Kit Scheduler, to support project enactment. Several clients are implemented to work on subtasks of the process. E.g. we developed a CAD/GIS client to work on the drawing tasks within the urban land-use planning process.

Figure 1 shows the system architecture of CoMo-Kit. It

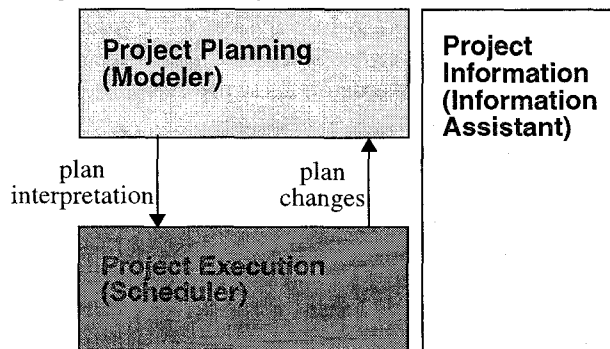


Figure 1: The Architecture of the CoMo-Kit System

consists of three main parts:

- The *Modeler* allows to plan a project.
- The *Scheduler* supports the execution of a project and manages the information produced.
- The *Information Assistant* allows to access the current state of a project.

In the following we explain the concepts behind the Modeler and the Scheduler. A description of the Information Assistant is omitted because it is not within the scope of this paper on „Project Coordination“.

3 Project Planning: The CoMo-Kit Modeler

Project planning means developing a model how the project should be carried out. Our approach uses ideas from KADS/CommonKADS [5] and applies them for project planning and coordination purposes.

For large-scale projects, a detailed plan can not be developed before the execution starts but planning and execution steps must be interlocked: Starting with an initial plan the first tasks are executed. Based on the results, the plan is refined and/or extended. Now we will define the ingredients of a project plan in our approach.

To model cooperative development processes, our Modeler uses four basic notions: Tasks (Processes), Methods, Products (Concepts) and Resources. In the following, these terms are defined as far as is necessary to understand this paper omitting the syntactical details of our project planning language.

3.1 Tasks (Processes)

A task describes a goal to be reached while working on a project. Our intended application domains deal with the production of information. Information which is relevant for the project is described in the project plan. Therefore we associate every task with a set of input and output¹ parameters². In the project plan, we are only able to state which type of information is used or must be produced.

Example: The task „requirements engineering“ uses an object of type „informal problem description“ as input and produces an object of type „requirements document“ as output.

For every input the following flags³ mentioned in Table 1 are defined.

Flag Name	Meaning
needed for planning	The input must be available before the planning of the task starts

Table 1: Parameter flags

1. If a parameter is modified in a task, it is modelled as input *and* output.
2. Parameters are also called “variables”.
3. The flags are mutually exclusive.

Flag Name	Meaning
needed for execution	The input is not needed for planning but it must be available before the execution starts.
optional	The input neither needed for planning nor for execution (but it may be helpful to have).

Table 1: Parameter flags

Outputs of a task may be optional or required.

Additionally, for every task a precondition and a postcondition may be defined. Preconditions are, for example, used to check if the inputs fulfill a given requirements. Postconditions are, for example, used to check if the output of a task has a wanted quality.

Example: For a task „implement program modules“ the precondition may be „all module specifications completed“ and the postcondition may be „module complexity < LIMIT“.

When tasks are executed, decisions are made which result in assignments of values to the output parameters. *Our approach assumes that there is a causal dependency between the available inputs and the produced outputs of a task.* We work on the principle that during project planning only inputs are associated to a task which are relevant and necessary for reaching the goal (Inadequacies of this assumption and their solutions are discussed in section 5.4).

3.2 Products (Concepts)

To model products which are created in the course of project execution, an object-centered approach is used. As usual, we distinguish between classes and instances. Classes define a set of slots to structure the product. Every slot is associated with its type and cardinality. Types may be other product classes or basic types (e.g. SYMBOL, STRING, REAL,...). During process enactment we represent product instances as values which are assigned to variables. The type of a variable is specified by a product class. Using other product classes as the type of a slot creates complex object structures. In the visualization of the product flows, these structures can be shown on different levels of detail.

3.3 Methods

To solve a task, a method is applied. For every task, there may exist a (predefined) set of alternative methods¹.

Further, our approach allows to define new methods when a task is planned or replanned.

Methods are executed by agents: they make a decision for one specific method depending on the context of the project.

We distinguish between atomic and complex (or composed) methods:

- Atomic methods assign values (= instances of product classes) to parameters. *Process scripts* describe how a given task is solved for humans. *Process programs* are specified in a formal language so that computers can solve a task automatically without human interaction.
- Complex methods are described by a product flow graph. A product flow graph consists of nodes which define parameters, (sub)tasks and links which relate parameters to tasks. The direction of the link determines if the parameter is the input or the output of the task. Every parameter is associated with a product type: During process execution, the parameters may store instances of this product type.

Subtasks can be further decomposed by methods. Along this line, the decomposition of the overall task (e.g. „develop software system“) can be seen as an AND-OR-Tree where tasks are AND-Nodes and methods are OR-Nodes. The appropriate method for solving a task is selected during process enactment.

3.4 Resources: Agents & Tools

Resources are used for project planning and task execution. *Agents* are active entities which use (passive) *tools* for their work.

Tasks are handled by agents. We distinguish between *actors* (= human agents) and *machines*.

For every task, the project plan defines the properties an agent must have to work on it. Further, our system stores information about the properties of every agent. For actors, we distinguish three kinds of properties: qualifications (q), roles (r), and organization (o).

During task execution, our system compares the required properties of a task with the properties an agent possesses. This allows to compute the set of agents which is able to solve the task.

Example: In a project plan, it is defined that the task „implement user interface“ should be executed by an actor which has skills in using the Visualworks Interface Builder (q), is a programmer (r), and works in department Dep 1.4 (o).

Having sketched our language for project planning, we now will explain how the execution of plans is supported.

1. For example, reusable plans may be extracted from old project traces and stored in an „experience factory“ [2]

4 Project Execution: The CoMo-Kit Scheduler

Conventional project management tools (e.g. MS Project) allow its users to plan a project and *manually* check if it advances as planned. The tool does not check if every planned task is executed nor does it provide the information which is necessary for task execution.

Conventional workflow management systems (cf. [13][16][17]) require a complete model to be available before the execution starts. Workflow management systems are only used for repetitive processes because the effort to develop a model is high. So, it is not feasible to use them for project planning.

Conventional groupware systems (e.g. Lotus Notes) basically are a means for communication and allow its users to access relevant information. They have no notion of a project plan and therefore are not able to support the coordination of projects.

On contrast, our Scheduler

- interprets the plan *automatically* and therefore is able to check the coherence between plan and execution¹. This coherence between the description of a process and its execution is required by quality management standards (ISO 9000 ff)
- is able to alternate planning and execution steps and develop a plan incrementally for a specific project (cf. [10])
- manages the tasks to be done and provides guidance for its users.

In the following, we will briefly illustrate the mode of operation of the Scheduler and its architecture.

4.1 The CoMo-Kit Scheduler

For every project, a new Scheduler Instance is created and initialized with a top level task. During task initialization, the task is delegated to a set of agents². Then the Scheduler waits until an agent logs in and accepts a task. After accepting a task, the agent chooses a method which decomposes it into several finer-grained tasks. This is a planning step and so the agent is only able to accept the task after all inputs which are required for planning are available. Then the agents delegates every subtask to a set of agents. If a task cannot be decomposed, the responsible agent has to produce the outputs. The Scheduler guarantees that an agent may only accept such a task if all needed inputs are available.

1. With the limitation that the system has to „believe“ a user who claims he is working on a task.
2. This set is a subset of the agents which are able to work on the task and are defined in the project model.

In Figure 2 the architecture of the CoMo-Kit Scheduler is shown.

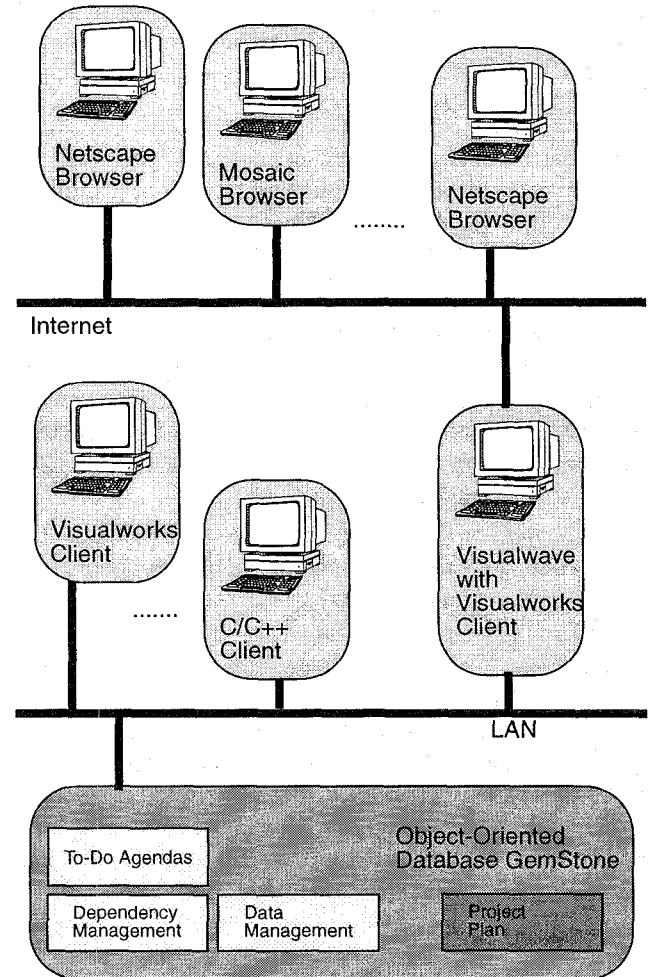


Figure 2 Architecture of the Scheduler

The server component of the Scheduler is implemented with the object-oriented database management system GemStone from Servio Corp. The server

- stores the current project model,
- handles the To-do agendas for every agent,
- stores all data produced during task execution³, and
- manages dependencies between project information (see below).

The server component is accessible via a local area network from Visualworks for Smalltalk-80 and C/C++-written clients. We developed clients⁴ in Visualworks which allow to

- accept tasks to work on
- plan a task
- change plans

3. If the data is produced using an external tool (e.g. Framemaker, CAD Systems etc.) then only a reference to the file is stored.
4. The user interfaces of these clients are shown in [20] or [22]

- decompose tasks into finer-grained subtasks
- supervise how the work on the subtasks is advancing
- edit products

For every task, a separate client can be used for planning. So, it is possible to distribute the overall planning process to several agents. Clearly, it is also possible to distribute the overall workflow (i.e. every of its subtask) to several agents.

Using the Visualwave Package from ParcPlace Systems it is possible to „forward“ the client interfaces via the world wide web (using Netscape, Mosaic or other Web-Browser as the front end to the user). So, Visualwave allows to distribute the work within a project via the Internet.

4.2 Developing & Executing Project Plans

Our framework allows to model arbitrary development processes. Anytime the current plan can be extended by adding new methods or changed¹. So, our approach supports incremental project planning.

From the point of view of a project manager, the planning and execution cycle is as follows:

- Define the initial task (or accept it from your boss).
- Describe the goal of the task, its inputs, outputs, and conditions.
- Execute the task *or*
- Decompose it into several subtasks and define the product flow between them.
 - Delegate the subtasks to your team members.
 - Supervise the subtasks execution.
 - React on exceptions by replanning or by notifying the producer of your input information or your boss.

To summarize the approach: The Scheduler supports the coordination of work processes by giving project managers means to plan a project and to supervise how the plan is followed.

A problem which remains is how to support large teams in reacting on changed decisions. In the following, we will show that if causal dependencies between decisions are handled by a computer-based system, change processes can be supported. The rest of this paper focuses on formalisms underlying the dependency management. Our approach allows to acquire these causal dependencies between decisions automatically from a project plan.

5 Project Changes: Improving Traceability

To manage causal dependencies between design decisions is necessary to guarantee the traceability of the development process. If a system is able to find out what is

1. If a project plan is extended or changed the system must check if the user has the appropriate access rights.

influenced by a decision, it is also able to inform its users if the decision is changed.

The dependency management component is based on ideas from REDUX [23][24]. Extending Petrie's approach, our system deduces dependencies automatically from a project plan. The postulated dependencies have to be made explicit and managed with a suited mechanism [11][21]. By automatically deducing dependencies from the project plan, the work load of users is reduced: They are not forced to enter all causal relations between decisions manually.

From a project plan our system derives two kind of dependencies:

- Product flow: The output of a task depend on the available inputs.
- Task decomposition: All subtasks of a task which becomes irrelevant have to become irrelevant too.

Now we will explain the formalisms which are used in our system to manage dependencies.

To solve a task, a *decision* is made which results in *variable assignments* or a *decomposition of a task into subtasks* (i. e. the selection of a decomposition method). *Rationales* are arguments for the validity of a decision or an information. Formally, a rationale is a logical implication between predicates.

During project execution, former decisions may be found erroneous or suboptimal. Then, they can be retracted and become invalid.

To formalize these notions, a set of predicates has been defined. At every point in time, a predicate can be *valid* (TRUE) or *invalid* (FALSE).

These rationales do not describe the discussion processes which lead to a decision. They are only a means to handle dependencies between decisions.

Whenever a decision *m* is made, the related predicate *decision(m)* is valid. If a decision must not be a part of the solution, the related method is *rejected*. The rejection of a method *m_i* is described by a predicate *rejected-decision(m_i)* and means that there is a hard reason against this decision. The relation between the two predicates is shown in Equation 1.

$$\text{rejected-decision}(m_i) \Rightarrow \neg \text{decision}(m_i) \quad (1)$$

For predicates, their rationales may be defined. Now we will explain how these rationales can be extracted automatically from the project plan.

5.1 Dependencies within the Information Flow

During the enactment of tasks, information is used and produced. The project plan specifies which information is used/produced and states the type of the information. *We assume that the information produced causally depends on*

the information used. This implies that the output information is in question whenever the inputs become invalid. An example illustrates the underlying formalism.

Figure 3 shows the information flow for a task T . I , O_1 and O_2 are formal parameters (variables) of products. They define the type of a product and give a name to a product instance within the process model. T consumes a product of type $PT1$ referenced by variable I . It produces products named O_1 and O_2 .

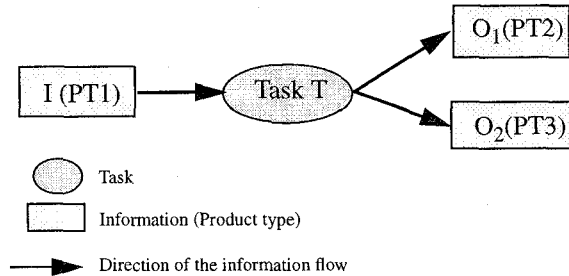


Figure 3 Information flow of a task T

Formally, the dependency between a decision and the assignment of values to the output parameters is expressed with equation 2. The validity of predicate $assignment(O_1=o_{1j})$ states that variable O_1 is assigned with value o_{1j} .

$$\begin{aligned} \text{decision}(m_j) \Rightarrow & \text{assignment}(O_1=o_{1j}) \wedge \\ & \text{assignment}(O_2=o_{2j}) \wedge \dots \wedge \\ & \text{assignment}(O_v=o_{vj}) \end{aligned} \quad (2)$$

If predicate $decision(m_j)$ becomes invalid, our system also invalidates the assignments of the output variables.

Our goal was to make the validity of the outputs dependent on the inputs. Formally, this is done by equation 3 together with equation 1.

$$\begin{aligned} \neg \text{assignment}(I_i=i_{1k}) \vee \dots \vee \neg \text{assignment}(I_n=i_{nk}) \\ \Rightarrow \text{rejected-decision}(m_j) \end{aligned} \quad (3)$$

For every task, equations 1-3 are created by our system as soon as a decision is made. Outputs of one tasks are inputs of others. During a design process, long chains of dependencies are created. These decision chains are the basis to guarantee traceability. *The equations state a general scheme for representing causal dependencies based on information flow.*

5.2 Dependencies between a Task and its Subtasks

Coordinating large projects requires that every person involved knows what to do and when to stop. If tasks are assigned to people and replanning takes place, some tasks

become invalid and the person working on it has to be informed about that fact. Our approach automates this and therefore reduces the project coordination effort.

Complex methods decompose tasks into several smaller subtasks. If during project enactment a decision for method m_i is made, then subtasks $T_1 \dots T_v$ become valid and must be solved. Formally, this dependency is expressed in equation 4. The predicate $validSubtask(T_i)$ is valid, as long as task T_i must be solved within the design process.

$$\begin{aligned} \text{decision}(m_i) \Rightarrow \\ \text{validSubtask}(T_1) \wedge \\ \text{validSubtask}(T_2) \wedge \dots \\ \text{validSubtask}(T_v) \end{aligned} \quad (4)$$

5.3 Dependencies based on the object structure

In CoMo-Kit, products are described in an object-centered manner. Therefore, there are dependencies between assignments which result from the object decomposition. E.g. if the assignment of a complex object to a variable becomes invalid, also all its slots (which are filled by other assignments) become invalid, too. These dependencies are implicitly contained in the equations above: Products are incrementally created during tasks enactment (by making a decision for a method). Every assignment of a value to a slot is part of the decision chain which starts from the creation of the object (no slots can be assigned if the object was not already created).

5.4 Limitations of the Automatic Deriveability of Causal Dependencies

The dependencies described above are automatically created based on the project plan. In practical applications, one has to live with incomplete project plans. This inadequacy may have several causes:

- A standard plan (e.g. the design process may be described in a quality management manual following ISO 9000ff) does often not match exactly to the actual project or may be too coarse-grained.
- Large projects often take a long period of time. In the beginning, these projects can only be roughly planned. For the first steps, perhaps a detailed plan may be generated. Later steps can only be planned in detail, when the results of the first are available (e.g. the implementation of a software system can only be planned in detail when a specification is available).

For the first point, one can argue that a “good” standard plan will not show this inadequacy. But the second point is inherent in the process and can *never* be solved by pre-planning.

These problems result in the possibility that a causal dependency between two data is not modeled in the project plan. It is also possible that the model shows an input information which is not relevant for a task. To overcome these two problems, our system allows its users to add new or delete unnecessary design rationales during task enactment.

Additional Rationales: Formally, new rationales are entered by changing equation 3. A decision may depend on:

- justifications based on the *validity* of previous decisions (cf. Eq 5)
- justifications based on the *validity* of existing parameter values (cf. Eq 6)
- justifications based on the *invalidity* of previous decisions (cf. Eq 7)
- justifications based on the *invalidity* of existing parameter values (cf. Eq 8)

The additional justifications change the condition for the rejection as demonstrated in the following formulas (Italic characters in following equations signal the additional justifications):

$$\begin{aligned} &\neg\text{assignment}(I_i=i_{1k}) \vee \dots \vee \neg\text{assignment}(I_n=i_{nk}) \vee \\ &\quad \neg\text{decision}(m_{old1}) \vee \dots \vee \neg\text{decision}(m_{oldM}) \\ &\Rightarrow \text{rejected-decision}(m_{new}) \end{aligned} \quad (5)$$

$$\begin{aligned} &\neg\text{assignment}(I_i=i_{1k}) \vee \dots \vee \neg\text{assignment}(I_n=i_{nk}) \vee \\ &\quad \neg\text{assignment}(X_I=x_{old1}) \vee \dots \vee \\ &\quad \neg\text{assignment}(X_M=x_{oldM}) \\ &\Rightarrow \text{rejected-decision}(m_{new}) \end{aligned} \quad (6)$$

$$\begin{aligned} &\neg\text{assignment}(I_i=i_{1k}) \vee \dots \vee \neg\text{assignment}(I_n=i_{nk}) \vee \\ &\quad \text{decision}(m_{old1}) \vee \dots \vee \text{decision}(m_{oldM}) \\ &\Rightarrow \text{rejected-decision}(m_{new}) \end{aligned} \quad (7)$$

$$\begin{aligned} &\neg\text{assignment}(I_i=i_{1k}) \vee \dots \vee \neg\text{assignment}(I_n=i_{nk}) \vee \\ &\quad \text{assignment}(X_I=x_{old1}) \vee \dots \vee \text{assignment}(X_M=x_{oldM}) \\ &\Rightarrow \text{rejected-decision}(m_{new}) \end{aligned} \quad (8)$$

To clarify the formulas: If a decision depends on a condition then, in our approach, its rejection depends on the negation of the condition.

Not necessary Rationales: Justifications of design decisions, which are automatically derived from the model, describe dependencies contained in the information flow. If some of those dependencies are irrelevant for a particular decision in the current project, it is necessary to delete them. As a result of this adaptation our system operates only on significant justifications.

We achieve the adaptation of the logical description by removing the related predicate (e.g. $\text{assignment}(I_n=i_{nk})$) of

a parameter assignment, that does not influence the decision. This causes the following change in Equation 3:

$$\begin{aligned} &\neg\text{assignment}(I_i=i_{1k}) \vee \dots \vee \neg\text{assignment}(I_{n-1}=i_{n-1k}) \\ &\Rightarrow \text{rejected-decision}(m_{new}) \end{aligned} \quad (9)$$

Of course, users are not forced to enter this formula directly but are supported by a graphical user interface.

To handle the dependencies, a truth maintenance system is used [12]

Now, we have explained the logical foundation of our work. Based on the dependency management techniques, our system is able to automatically inform involved users about changes.

In a local area network, this forward propagation is easily implemented using the mechanisms of the GemStone-Smalltalk Interface. In the world wide web, the forward pushing of information from the server to the client is a bit unconventional. Alternatively, the users may manually update their task agenda by accessing the server from time to time.

6 State of Implementation

The CoMo-Kit Modeler is fully implemented as well as the CoMo-Kit Scheduler for the local area network. The integration of CoMo-Kit with the world wide web, using the Visualwave package, currently allows for plan execution and method selection/rejection. Defining new methods currently is not supported via the Web.

7 Summary & Future Work

The CoMo-Kit approach allows to alternate planning and execution of design processes. One main focus of the work is to handle dependencies between design decisions to improve project traceability and support humans in reacting on changes in a project. In our opinion, the techniques developed so far are necessary for supporting and coordinating (globally or locally) distributed workflows.

Our approach addresses the basic questions mentioned in section 1

- Who should do what?

The model describes the project plan and answers this question by describing the needed properties of an agent who will be able to work on a task. Additionally, the manager of a task is able to delegate its subtasks to specific agents.

- When can the work on a task start?

The precondition and the flags of a task allow to define when the planning and the execution on it can be started.

Thereby, for example, it is possible to check if all needed inputs are available.

- Who must be informed about changes?

The Scheduler is based on a generic, reusable model of change propagation: It sends messages to the appropriate users when an information item becomes invalid and it also notifies users that a task has become invalid. Clearly, this should be extended by domain specific change propagation strategies and user-specified change requests (cf. [19]).

CoMo-Kit follows a centralized, black-box-agent approach (here we use the term "agent" in the sense of Distributed AI and *not* as we defined it in Section 3.4). Every agent, as for example the Scheduler, has a location in the network and may cooperate with other agents distributed in the net. In contrast, [1] and [4] have white box agents which allow for a higher flexibility in the agent structure itself as well as in the cooperation strategies. On the other hand, this flexibility has to be paid for because the functionality of the CoMo-Kit Scheduler has to be reimplemented when using these approaches.

Based on our theoretical work, we implemented the CoMo-Kit Scheduler which allows to plan, coordinate and execute design processes.

We apply CoMo-Kit in two application domains: urban land-use planning [20] and software engineering. [3] and [7] also deal with SE. The first describes problems resulting from virtual cooperations, the second uses WWW techniques to support software development and metrics collection.

In future, we will integrate constraint techniques into the server component to handle dependencies between decisions which are not directed (see [15] for similar work). Additionally, we will use techniques from [14] and [8] to improve CoMo-Kit concerning basic project management and scheduling techniques.

8 References

- [1] M. Barbuceanu, Mark S. Fox: Coordinating Multiple Agents in the Supply Chain, in: [27]
- [2] V. R. Basili: The Experience Factory and its Relationship to Other Improvement Paradigms, in: Ian Sommerville, Manfred Paul (eds.): Proc. of the 4th European Software Engineering Conference, P. 68-83, Lecture Notes in Computer Science Nr. 706, Springer Verlag, 1993.
- [3] C. Boldyreff, J. Newman, J. Taramaa: Managing Process Improvement in Virtual Software Corporations, in: [27]
- [4] F. M. T. Brazier, C. M. Jonker, J. Treur: Modeling Project Coordination in a Multi-Agent Framework, in: [27].
- [5] J. Breuker, W. van de Velde (eds.): CommonKADS Library for Expertise Modeling, IOS Press, 1994.
- [6] S. Buckingham Shum and N. Hammond. Argumentation-based design rationale: what use at what cost? *Int. J. Human-Computer Studies*, 40:603-652, 1994.
- [7] J. Callahan, S. Ramakrishnan, W. Sun: Software Project management and Measurement on the World-Wide-Web, in: [27]
- [8] K.J. Cleetus, C. Cascaval, K. Matsuaki: PACT - A Software Package to Manage Projects and Coordinate People, in: [27]
- [9] J. Conklin, Begeman: gIBIS: A Tool for Exploratory policy Discussion, In *Proceedings of CSCW '88 and ACM Transactions on Office Information Systems*, October, 1988.
- [10] B. Dellen, F. Maurer: Integrating Planning and Execution in Software Development Processes, in: [27].
- [11] B. Dellen, F. Maurer, J. Paulokat: Verwaltung von Abhängigkeiten in kooperativen, wissensbasierten Arbeitsabläufen. In *Richter, Maurer (Hrsg.): Expertensysteme 95*, pages 72-89, infix Verlag, 1995
- [12] J. Doyle: A Truth Maintenance System, *Artificial Intelligence*, 12:231-272, 1979.
- [13] D. Georgakopoulos, M. Hornick: An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure, *Distributed and Parallel Databases*, 3, pages 119-153, Kluwer Academic Publishers, Boston 1995
- [14] S. Goldmann: Procura: A Project Management Model of Concurrent Planning and Design, in: [27]
- [15] L. Gupta, John Chionglo, Mark Fox: A Constraint-Based Model of Communication and Coordination in Concurrent Design Projects, in: [27]
- [16] S. Jablonski: Workflow-Management-Systeme: Motivation, Modellierung, Architektur: From Informatik Spektrum 18: pages 13-24, Springer-Verlag 1995
- [17] A. Oberweis: Verteilte betriebliche Abläufe und komplexe Objektstrukturen: Integriertes Modellierungskonzept für Workflow-Managementsysteme, Habilitation Universität Karlsruhe, 1994
- [18] A. Mac-Lean et al.. Questions, Options and Criteria; Elements of Design Space Analysis. *Human-Computer Interaction*, 6: 201-250, 1991.
- [19] F. Maurer: Computer Support in Project Coordination, in: [27].
- [20] F. Maurer, G. Pews: Supporting Cooperative Work in Urban Land-Use Planning, Proc. COOP-96, 1996.
- [21] F. Maurer, J. Paulokat: 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.
- [22] F. Maurer, G. Pews: Ein Knowledge-Engineering-Ansatz für kooperatives Design am Beispiel der Bebauungsplanung, Themenheft Knowledge Engineering, KI 1/95, interdata Verlag, 1995.
- [23] Ch. Petrie: Planning and Replanning with Reason Maintenance, Dissertation, University of Texas, Austin, 1991.
- [24] Ch. Petrie, M. Cutkosky. Design space navigation as a collaborative aid. In J. Gero and F. Sundweeks, editor, *Artificial Intelligence in Design '94*, Kluwer Academic Publishers, 1994
- [25] Ch. Petrie: The Redux' Server, Proc. Internat. Conf. on Intelligent and Cooperative Information Systems (ICICIS), Rotterdam, May, 1993.
- [26] C. Potts and G. Bruns, Recording the Reasons for Design Decisions, In *Proceedings of the 10th International Conference on Software Engineering*, pages 418-427, 1988.
- [27] Proceedings of WET ICE 96, IEEE press, 1996
- [28] Special Issue on Design Rationale, *Human-Computer Interaction*, 6: 197-419, 1991.