Enriching Software Process Support by Knowledge-based Techniques[*]

Barbara Dellen, Frank Maurer, Jürgen Münch, Martin Verlage[+]

Fachbereich Informatik, Universität Kaiserslautern, Postfach 3049, 67653 Kaiserslautern, Germany
{verlage, dellen, maurer, muench}@informatik.uni-kl.de

Abstract. *Representations of activities dealing with the development or maintenance of software are called software process models. Process models allow for communication, reasoning, guidance, improvement, and automation. Two approaches for modeling processes and instantiating and managing the process models, namely CoMo-Kit and MVP-E, are combined to build a more powerful one. CoMo-Kit is based on AI/KE technology; it is a support tool system for general complex design processes, and was not been developed specifically with software development processes in mind. MVP-E is a process-sensitive software engineering environment for modeling and analyzing software development processes, and guides software developers. Additionally, it provides services to establish and run measurement programmes in software organizations. Because both approaches were developed independently from one another, major integration efforts had to be made to combine their both advantages. This article concentrates on the resulting language concepts, and their operationalization necessary for building automated process support.*

1 Introduction

Processes are present whenever information is created, transformed, or communicated. Although we distinguish between different types of processes (e.g., business processes, decision processes, and software development processes) these types share common properties. Understanding similarities and differences between process types is a key factor for better process support. Process support includes improved communication, detailed reasoning about process features, guiding people when performing processes, improving both the process itself and its results, and automating process steps in order to gain deterministic process behavior [10, 36]. These purposes require explicit process representations (i.e., *process models*).

[*]This work is supported by the Deutsche Forschungsgemeinschaft as part of the Sonderforschungsbereich 501 “Development of Large Systems with generic Methods”. See also the explanation in the appendix.
[+] In alphabetical order.

The variety of existing process support systems, for example process-sensitive software engineering environments or workflow management systems, corresponds to the variety of process types. In this article we focus on software development processes and software processes. A *software development process* is a (sub-)set of the technical activities to build a *software product*. Maintenance tasks are subsumed by this phrase. A *software process* includes all software development processes and all organizational processes needed to drive the project (e.g., management processes). The addition 'software' may be omitted in this article for the sake of brevity. Interpretation of development process models, performed by a *process engine*, is called *enactment* to emphasize the necessity of user participation for process performance [15]. A process engine commonly is part of a so-called *process-sensitive* or *process-centered* software engineering environment [17].

Process-sensitive software engineering environments use process models to provide sophisticated support for both software developers and organizational roles [17]. Enaction updates a represented of the process state in order to reflect the real-world processes accurately. This works quite well if the process behaves as planned. Nevertheless, from time to time events may occur which had not been considered when modeling the process. The real-world process and its model do not match any longer. *Process engines* should provide mechanisms for replanning a project under enaction. This addresses the problem of *process evolution* [27]. Without these features such an environment is not applicable in real-world projects. Unfortunately, only limited solutions do exist.[1]

Providing such a powerful process-sensitive software engineering environment is a major goal of our research work. Two already existing approaches from knowledge engineering and software engineering domains are synthesized. The Conceptual Model Construction Kit (CoMo-Kit) was developed for supporting complex design processes (e.g., city planning). The Multi-View Process Environment (MVP-E) supports the modeling and analyzing of software development processes, and guides software developers. Additionally it provides services to establish and run measurement programmes in software organizations. Both research prototypes were developed independently from one another. Suitability of both approaches to real problems was demonstrated [22,31]. By studying the approaches, relating their underlying assumptions, and synthesizing them we gain a better understanding about the principles of process support environments. The goal is to provide knowledge-based technology for software engineering problems. The synthesized approach should allow to prescribe development processes without hindering creativity. Necessary restrictions and rules delineate human process performance. It is intrinsic to software development processes that at any time in the project the next steps can be described precisely, whereas the later steps do not have a sharp shape. Process support environments should provide mechanisms to adjust the project plan from time to time. Moreover, capturing dependencies during product evolution allows goal-directed backtracking in order to bring the project back to the right track. Although this vision is still to be achieved, promising results of synthesizing CoMo-Kit and MVP-E already exist. They concentrate primarily on issues relevant for developers rather than providing support for project managers (e.g., resource scheduling).

1."...there is much further research work to be carried out in the area of software process evolution. Understanding and managing software process evolution seems to be one of the most difficult challenges that the software engineering community is facing today." ([27], p. 1126)

As a first step of integration a careful analysis revealed commonalities and differences of CoMo-Kit and MVP-E. This was used to define requirements for process-sensitive software engineering environments from our particular perspective [40]. The integration of CoMo-Kit and MVP-E requires the identification of similar concepts. It is interesting that the concepts of CoMo-Kit are a subset of those present in MVP-L; this is a sign that both approaches contain concepts that are intrinsic to processes in general. On the other hand, the functionality of the CoMo-Kit process engine exceeds MVP-E because it allows modeling, planning, and enaction steps to alternate and supports change processes.

In this article, we discuss the next two steps of integration, the definition of a common process representation schema or language and the implementation of a prototypical process engine. The core of the new language, which is called *Modeling Language and Operational Support for Software Processes* (MILOS), is operationalized by an integrated process engine. In this article, special attention is paid to the traceability of the defined requirements set up for the process engine. The collaboration between CoMo-Kit and MVP-E is illustrated using a scenario which describes replanning a process under enactment:

The scenario embodies a model of a standard implementation process, as for example described in [20]. A module's design is complete, and source code is to be created. In parallel, test data has to be derived either by analyzing the code (in the case of a later structural or white-box testing) or by eliciting the data from the requirements document (in the case of a later functional or black-box test). The choice of selecting the first or second alternative of test data derivation depends on the module's control flow complexity. If complexity is high then structural testing is applied, else functional testing should be selected. Because the module's complexity is not known prior to its design or implementation (depending on the complexity measure) resources cannot be assigned to the data derivation process. As soon as the value is measured, project management instantiates the corresponding process, and schedules a responsible developer for it. Please note that although this could be described as a simple if-then-else situation in a process script, the task for a process management system is much more difficult. Planning makes statements - sometimes in a vague manner - about future objects. The more the project proceeds the more concrete statements can be made. In the above example, derivation of test data for either functional or structural testing is allowed to be instantiated. One reason is that resources should be assigned only to one of these two tasks. Moreover the process engine must manage further refinement of already instantiated processes (e.g., insert a refinement of derivation of test data for functional testing which includes equivalence or boundary analysis) and retracting decisions already made (e.g., the system's design is modified and the module is split into two modules).

We understand the work presented in this article as an example of how knowledge-based techniques can be used to address one of the major problems in the area of automated support for managing software development processes.

The article is organized as follows: The concepts implemented in MVP-E and CoMo-Kit are surveyed in Sections 2 and 3. Section 4 identifies requirements for process support environments. They are derived from a comparison of both approaches in [40]. The integration is described in the Sections 5-8: Section 5 explains the integrated system architecture, Section 6 discusses concepts of the integrated language, Section 7 gives an example, and Section 8 outlines the operationalization of the language concepts. In Section 9 we compare our findings to related research work. Finally, Section 10 summarizes the article and gives an outlook on future work.

2 MVP-E

The MVP project aims at support for management of software development processes from a software engineering perspective. Properties of such processes are, for example:

- Many people are involved in a project and perform many different types of processes.
- The processes last long, sometimes several months or years.
- Not all process steps are known in advance when planning the project.
- Erroneous performance or bad process models require repeated performance, probably also of other processes than the failed one.

The MVP project began at the University of Maryland and continues at Universität Kaiserslautern, Germany. Its goal is to develop MVP-E, as an instance of the ideas developed in the TAME project [6]. Special attention is paid to measurement activities which are essential if the environment is to be used by all roles of the project and the organization [26]. MVP-E supports modeling, planning, simulation, enactment, measurement, recording, and packaging of software engineering processes. The main idea is to split descriptions of software development processes into views. Every view is generally defined as a projection of a process model that focuses on selected features of the process [39].

The language MVP-L (multi-view process modeling language) is used to describe development processes. It distinguishes between processes, products, resources, and their attributes which correspond to measurable qualities of the objects; processes, products, and resources are instantiated with respect to types, and are put into relation in a project plan [8]. A second notation is used to represent GQM trees which are specifications of measurement goals from a specific viewpoint [5]. The discussion of GQM is beyond the scope of this article. MVP-L has been evaluated in several industrial settings (e.g., [22]). The case studies' feedback became input for the evolution of MVP-L. Table 1 summarizes the main concepts of MVP-L.

Concept	Explanation
Process	Activities which create, modify, or use a product.
Product	Software artifact describing the system to be delivered.
Resource	Human agent or tool.
Attribute	Measurable characteristic of a process, product, or resource.
Criteria (entry, invariant, exit)	Expression which must be true when starting, enacting, or terminating a process respectively.
Refinement	Breaking down the structure of a process, product, or resource into less complex parts.
Instantiation	Creating an instance of a process type, product type, or resource type and providing actual parameters.
Product Flow	Relationship between processes and products. It is distinguished between reading, writing, and modifying access.
Resource Allocation	Assigning personnel to processes in order to perform the processes.
Process Model, Product Model, Resource Model	Type descriptions of processes, products, and resources respectively. The name *Model* might be misleading in this context, because their instances are models of real-world processes, too.

Table 1: Concepts of MVP-L

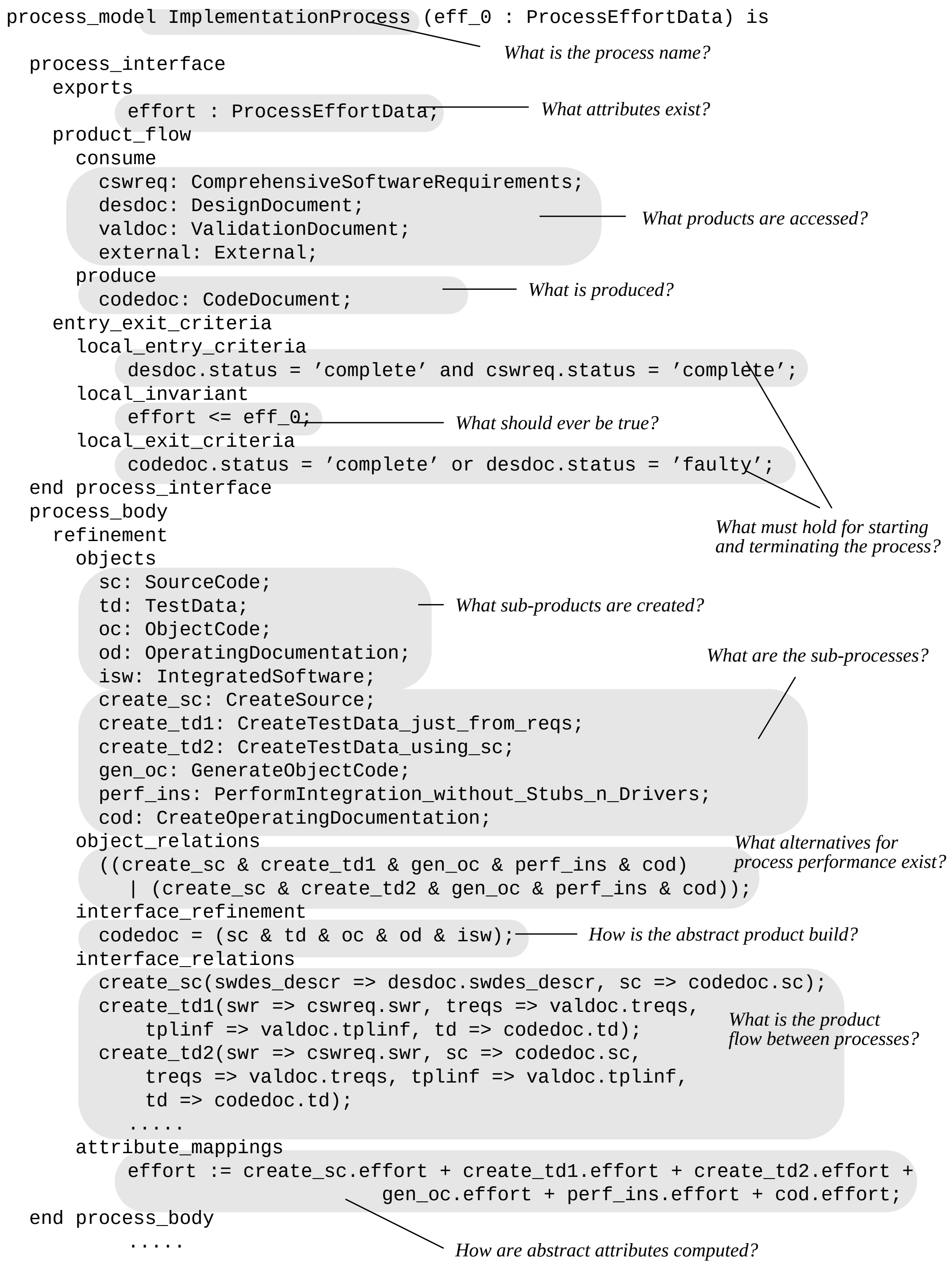
Figure 1 shows an excerpt of an MVP-L example from the scenario above. It stems from a formalized standard implementation process defined in the IEEE Standard 1047-1991 [20]. Since the implementation process can be seen as a typical process in a software life cycle, it is used throughout the whole article to demonstrate the modeling styles of the different approaches. For the purpose of a comprehensible illustration, minor differences among the process descriptions in the standard and the adapted examples were accepted.

MVP-E's process engine MVP-S uses a project plan and process models (types) to build its own representation of a real-world project [26]. The process engine is used to manage project data and to guide developers in their work. Processes can be enacted if they are *enabled* (i.e., their entry criteria are true). The agents, represented as resources assigned to processes, are responsible for achieving the goals specified in the exit criteria, without invalidating the process invariants. Throughout enactment, measurement data is taken which is used by all project roles (e.g., testers, managers) to reason about the project and to trigger actions. The following assumptions (labeled as **AM*i***) were made during the evolution of the MVP-E approach and are made explicit to allow the comparison of the underlying motivations and goals with those of CoMo-Kit later on:

AM1: The concepts implemented in MVP-L are suited to express a single view on a particular software process and are made explicit to allow to compare the goals and motivation of different approaches.

AM2: Attributes of processes, products, and resources are sufficient for all measurement purposes.

AM3: Developers are guided by enacting understandable process models.



```
process_model ImplementationProcess (eff_0 : ProcessEffortData) is

  process_interface
    exports
          effort : ProcessEffortData;
    product_flow
      consume
        cswreq: ComprehensiveSoftwareRequirements;
        desdoc: DesignDocument;
        valdoc: ValidationDocument;
        external: External;
      produce
        codedoc: CodeDocument;
    entry_exit_criteria
      local_entry_criteria
          desdoc.status = 'complete' and cswreq.status = 'complete';
      local_invariant
          effort <= eff_0;
      local_exit_criteria
          codedoc.status = 'complete' or desdoc.status = 'faulty';
  end process_interface
  process_body
    refinement
      objects
        sc: SourceCode;
        td: TestData;
        oc: ObjectCode;
        od: OperatingDocumentation;
        isw: IntegratedSoftware;
        create_sc: CreateSource;
        create_td1: CreateTestData_just_from_reqs;
        create_td2: CreateTestData_using_sc;
        gen_oc: GenerateObjectCode;
        perf_ins: PerformIntegration_without_Stubs_n_Drivers;
        cod: CreateOperatingDocumentation;
      object_relations
        ((create_sc & create_td1 & gen_oc & perf_ins & cod)
           | (create_sc & create_td2 & gen_oc & perf_ins & cod));
      interface_refinement
        codedoc = (sc & td & oc & od & isw);
      interface_relations
        create_sc(swdes_descr => desdoc.swdes_descr, sc => codedoc.sc);
        create_td1(swr => cswreq.swr, treqs => valdoc.treqs,
            tplinf => valdoc.tplinf, td => codedoc.td);
        create_td2(swr => cswreq.swr, sc => codedoc.sc,
            treqs => valdoc.treqs, tplinf => valdoc.tplinf,
            td => codedoc.td);
          .....
      attribute_mappings
          effort := create_sc.effort + create_td1.effort + create_td2.effort +
                                 gen_oc.effort + perf_ins.effort + cod.effort;
  end process_body
          .....
```

What is the process name?
What attributes exist?
What products are accessed?
What is produced?
What should ever be true?
What must hold for starting and terminating the process?
What sub-products are created?
What are the sub-processes?
What alternatives for process performance exist?
How is the abstract product build?
What is the product flow between processes?
How are abstract attributes computed?

Fig. 1: MVP-L example of an implementation process with two alternatives

AM4: All project roles are supported by using a common representation of a project. Each role gets its own views on the project which are tailored to its specific needs.

AM5: Products are represented without storing their contents. Direct access to the product is not supported. Only product models are accessed.

AM6: The steps of modeling, planning and enaction are sequential. Project plans remain unchanged over the project lifetime.

AM7: Planning provides parameters which adapt process models to different contexts.

AM8: Plan deviations are not considered. Support for modifying the project's state is not provided.

Several other approaches for supporting software development processes have been developed [1, 10, 36]. In general, we can distinguish between languages for modeling fine-grain processes (i.e., used to integrate tools, hence being similar to programming languages) and coarse-grain processes (i.e., used to guide software developers and to coordinate their tasks, hence being similar to specification languages). Currently, the software engineering community focuses on the latter kind of processes. An example is the evolution of the Marvel Strategy Language, which was first designed for use in a single-user system and now supports distributed teams [7]. MVP-L belongs to the second category of languages, too. Another classification of software process languages can be made with respect to their level of abstraction. Granularity ranges from abstract levels with rich semantics of the concepts (e.g., MVP-L) down to detailed levels which provide powerful mechanisms to build one's own process building blocks (e.g., APPL/A [37]).

Not all languages cover an equal set of aspects of software development processes. Mostly, the languages provide a solution for a particular problem and support only a limited set of roles [35]. For example, the language SLANG focuses on capturing the dynamic aspects of a process [3]. MVP-L addresses process interfaces, rule-based specification of development processes and measurement aspects. Quality considerations have become more and more important within software development. However measurement aspects have not been taken into account in most of the process support environments. Approaches to support measurement activities within software development already exist [24]. MVP-E is an environment which includes measurement support.

3 CoMo-Kit

The CoMo-Kit project at Universität Kaiserslautern aims at support for planning, enacting, and controlling complex workflows in design processes from a knowledge engineering perspective [28]. Critical properties of such workflows are:

- They are too complex to be planned in detail before enactment starts. Results of activities are needed to plan later steps. Planning and enacting must alternate during the whole project.

- Decisions are made during enactment which rely on assumptions that can be proved invalid afterwards. When decisions must be rejected, users must be supported in reacting to the new situation (i.e., backtracking must be supported).

The CoMo-Kit project tackles the problems that are caused by these properties. The CoMo-Kit system consists of a tool which allows to model complex workflows, and a workflow management system, the CoMo-Kit Scheduler, which enacts and controls the modeled workflows. To describe complex design processes, the CoMo-Kit methodology uses four basic concepts: tasks, methods, concepts, and agents. Table 2 contains an overview and short descriptions of the terms. Using these concepts generic software process models and concrete project plans can be described. The enactment of design processes is supported by a flexible workflow management system, the CoMo-Kit Scheduler. In Table 3, concepts of the Scheduler are explained. The main features of the Scheduler are:[1]

- The Scheduler allows for alternation of planning and enactment of development processes.
- Based on the information flow between tasks, causal dependencies emerge during enactment. The underlying assumption is that the inputs of a task influence the outputs. For every task a set of logical implications is created; the implications relate the assignment of values (i.e., products) to the input variables of a task with the assignment of values to the output variables. Whenever the assignment of an input variable becomes invalid, the assignments of the output variables become invalid too. The causal dependencies improve the traceability of design decisions and support the users in reacting to changes.
- The scheduler manages dependencies extractable from the task decomposition. Whenever a task is invalidated by replanning of the project, the Scheduler notifies team members working on subtasks that they can stop working on them.

1.The techniques which are used to implement these features are beyond the scope of this article. For a detailed description see [29,30]. For a description from a software engineering point of view see [11].

• To handle dependencies efficiently, reason maintenance techniques are used [33,34] and extended. The Scheduler uses the justification structures to support dependency-directed back-tracking.

Concept	Explanation
Task (Process)	A description of the goal which should be reached by an activity.
Input Variables	Information which is needed to work on a task.
Output Variables	Information which is the outcome of working on a task.
Method	A description of how a task’s goal can be reached. For every task a set of alternative methods can be described.
Atomic Method	Atomic methods assign values to the output variables of the related task.
Complex Method	Complex methods decompose a task into subtasks.
Information Flow	A complex method is described by an information flow graph which consists of (sub-) tasks and variables. The information graph shows the input/output relations of tasks.
Concept Class	A description of the structure of the information (product structure) which is produced during enactment.
Slot	Stores part of product information.
Concept Instance	Concrete information, for instance a product (such as requirements document) which is the outcome of using a method to solve a task.
Agent	An actor who works on tasks. Agents apply methods to solve tasks.

Table 2: The CoMo-Kit modeling framework

The following assumptions (labeled as **AC*i***) were made during the evolution of the CoMo-Kit approach:

AC1: Modeling, planning, and enacting design processes are interfering steps and often alternate.

AC2: Project plans have to be modified and refined during project lifetime.

AC3: The Scheduler manages the state of the project. Therefore, it should have access to all products and manage them.

AC4: Users must be supported in tracking of where the result of their work is used and what information they need in producing their results.

AC5: Changing decisions is inherently needed in complex projects. This includes the management of their effects (i.e., changes of project states).

AC6: Causal dependencies are the basis for an active notification mechanism which informs users about relevant changes of the project state.

AC7: Only processes performed by individuals are modeled in order to keep track of dependencies.

AC8: One formalism can be used to describe both generic process models and concrete project plans.

Concept	Explanation
Decision	To solve a task (i.e., to reach the goal) an agent has to decide which method should be applied.
Assignment	Applying an atomic method to a task results in the assignment of values to its output variables.
Task Decomposition	Applying complex methods to tasks results in a set of subtasks which are included into an agenda. This agenda stores a list of all tasks which must be solved to finish the project.
Dependency	From the information flow a set of causal dependencies is derived. We assume that there is a causal dependency between the inputs of a task and its outputs. The Scheduler manages these dependencies.
Decision Retraction	During project enactment decisions can be found erroneous, which results in an inconsistent project state. Then, at least one decision must be retracted and replaced by another alternative.

Table 3: Concepts of the CoMo-Kit Scheduler

Figure 2 shows the process decomposition from the scenario of Section 1 modeled in CoMo-Kit. The product flow within the method “Implementation with Structural Testing” is shown in Figure 3.

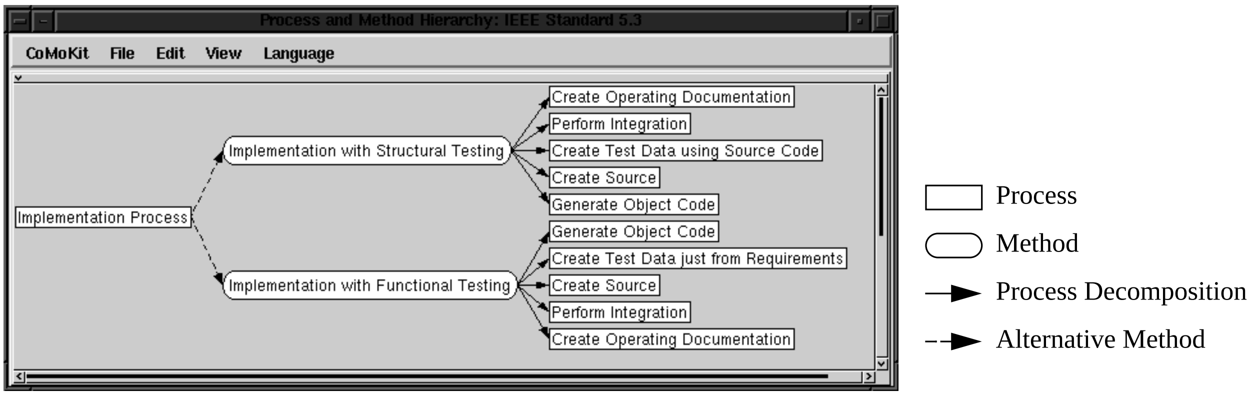

Process and Method Hierarchy: IEEE Standard 5.3
CoMoKit
File
Edit
View
Language
Implementation Process
Implementation with Structural Testing
Implementation with Functional Testing
Create Operating Documentation
Perform Integration
Create Test Data using Source Code
Create Source
Generate Object Code
Generate Object Code
Create Test Data just from Requirements
Create Source
Perform Integration
Create Operating Documentation
Process
Method
Process Decomposition
Alternative Method

Fig. 2: Process decomposition in CoMo-Kit

Several other approaches for supporting workflows have been developed. In [18] an overview on current workflow management techniques is given: Current technology assumes that the process model is defined before process enactment. The CoMo-Kit approach was developed for application domains where planning and enactment alternate, for example design processes (e.g., city planning). Some workflow management systems support object-oriented data structures. For instance in [32] an innovative approach is described which integrates Petri nets with semantic

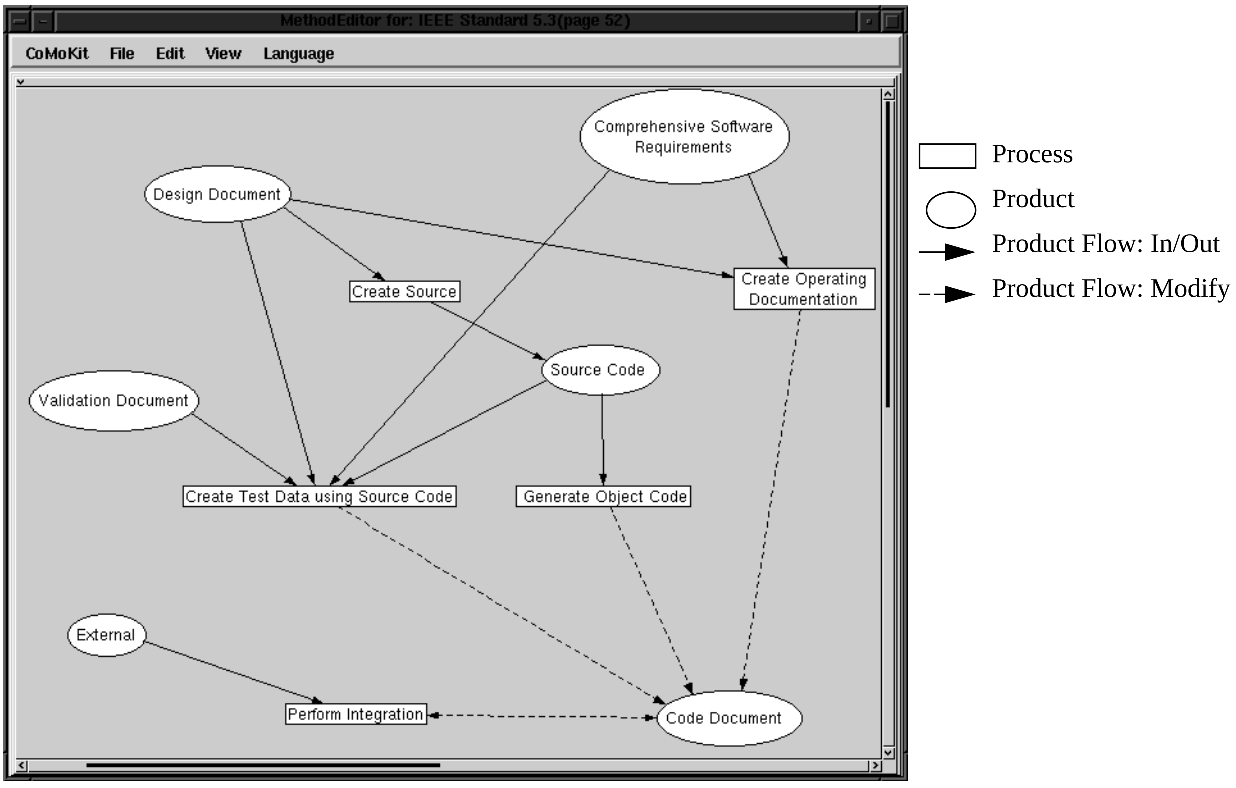


Fig. 3: Product flow in CoMo-Kit

data models. CoMo-Kit also supports object-oriented data structures. In addition, CoMo-Kit supports reacting to decision changes.

4 Requirements

Section 2 and Section 3 presented assumptions which guided the evolution of two process support approaches. The assumptions can be regarded as requirements for a process-sensitive software engineering environment. The following list of requirements for process support environments reformulates the assumptions made in both approaches. Incompatibilities caused by different assumptions in the two approaches are removed (as shown in [40]), so the requirements represent a unified view. For each requirement (labeled as **R*i***) the related assumption(s) (AM*i* or AC*i*) are given. The set of requirements is not to be understood as an exhaustive list of features a process-sensitive software engineering environment should have. The requirements were defined from the perspective of both systems CoMo-Kit and MVP-E. Nevertheless we consider them as mandatory for any support environment for real development processes. Some aspects are not discussed in this article (e.g., resource scheduling, time planning, budgeting).

R1: *The main concepts of software development processes should be provided.* (AM1, AC7, AC8) Algorithms for planning, analyses, or enactment need to implement the semantics of basic entities (e.g., processes, products, product flow). It is important to tailor the concepts' representation to human and machine understanding.

R2: *Organizational processes should be supported.* (AM4) Specific interfaces to organizational processes are required. Further development in the area of multiple views on a project representation is needed in order to gain role-specific representations [39].

R3: *Both short-term planning (detailed level) and long-term-planning (abstract level) have to be supported* (AM7, AC1). It is desirable to plan activities early in a project, but it is not practical in every case. Mechanisms are needed for planning abstract and general processes as well as for planning concrete and state-dependent ones.

R4: *Allow for alternating modeling, planning and enaction.* (AM6, AC1, AC2) Information for planning is incomplete when launching the project. Required information may be produced during the project. Later planning steps refine the models or the original plan.

R5: *Document decisions.* (AC4, AC6) Many decisions about products and processes need to be documented [37]. The decisions explain how products evolve and processes are performed. The decisions are used to recognize deviations and to notify the system of inconsistencies. Dependencies between decisions can be used for backtracking and also for reasoning about products [11].

R6: *Reactions to changing decisions.* (AC5) When an invalid state is reached after retracting a decision, backtracking should be performed in order to reach a branch where an alternative can be chosen.

R7: *Manage the products whenever possible.* (AM5, AC3) Inconsistencies between the real-world project and the representation managed by the environment must be avoided.

R8: *Support measurement activities.* (AM2) Quantitative data from processes and products are needed in order to develop software systematically [36]. Measurement, evaluation, and storage of data is needed.

R9: *Software developers need to be guided.* (AM3, AC4) Process models explain what activities to perform next and what their goals are.

R10: *Software developers' tasks need to be coordinated.* (AM4, AC6) Process models are used to relate tasks of software developers. In the case a result becomes invalid, coordination means notifying others to interrupt their work.

R11: *Execution of process fragments.* Process programs are fine-grain process models. They are described in a formalism similar to programming languages. Process programs are used to automate process steps. It must be ensured that the dependencies established during a tool invocation are captured and documented in the environment's repository.

The first ten requirements describe features of either CoMo-Kit or MVP-E. Automatic execution of process programs is necessary. This functionality should be to realize without bigger problems because experience from other projects exists [36]. Commercially available process-sensitive software engineering environments (e.g. Process Weaver [16]) have already demonstrated the suitability of a shell-like language for the definition of process programs. Fulfilling all mentioned requirements in the system is the challenge of integrating CoMo-Kit and MVP-E. Nevertheless, there are still open problems which are not tackled by the synthesis of both approaches:

P1: *State Manipulation.* If a project deviates from the plan it might be necessary to shift the whole project state by a "brute-force" manipulation of state variables instead of performing backtracking. The problem is how to modify the models according to the new state of the real-world project and to preserve existing dependencies.

P2: *Type-Instance Correspondence.* Direct manipulation of instances results in a project trace which cannot be described by the plan's process types. This is also true when types of active processes are modified. Mechanisms are needed to establish correspondence between instances and types.

P3: *Existing Products and Measurement Data.* Backtracking and choosing another alternative should not mean throwing away the data already measured and the products already developed. Mechanisms for reuse within a project are needed. At the moment it is not clear how to handle measurement data in the case of backtracking. For example, effort data should be kept but statements about products (e.g., subjective classification of complexity) might become invalid.

Although the requirements listed in this section present a unified view of both approaches this does not mean that a particular user has to handle such a tool in its entire complexity. Some features might be interesting for only some roles (e.g., a project planner is interested in R4 but not in R9) and some functionality should be kept completely away from users (e.g., R10).

5 The Integrated System Architecture

Within the CoMo-Kit project techniques, methods, and systems were developed which support planning and enacting complex distributed cooperative design processes [12,31]. This system is the basis for the prototypical implementation of an integrated system, called the MILOS environment, which fulfills requirements R1-R11.

Figure 4 shows the system architecture of MILOS. It consists of three main parts:

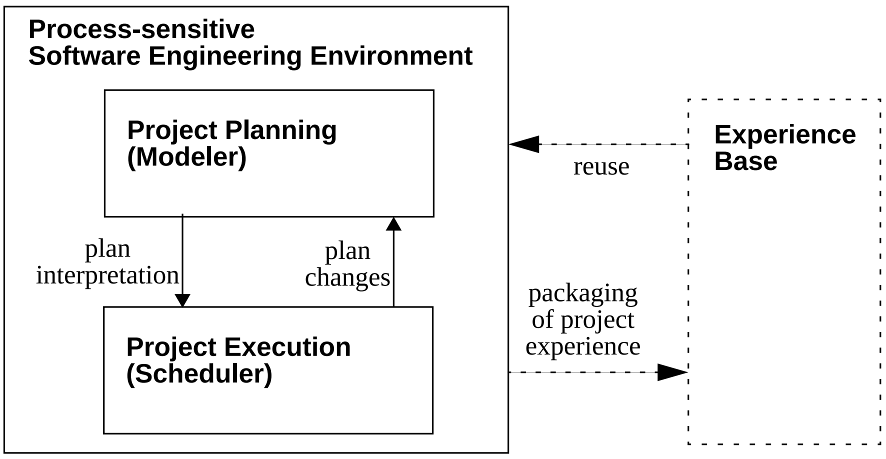


Fig. 4: The Architecture of the MILOS System

- The *Modeler* allows to plan a project and to extend and modify that plan.
- The *Scheduler* supports the enactment of a project plan and manages the project state and history (i.e., the project track).
- The *Experience Base* stores generic (reusable) project plans, products etc.

The MILOS system is organized as a proactive client/server application. The MILOS Scheduler as the project plan server gets the types from the modeler, instantiates them and manages the current project state. It informs the clients in order to guide software developers. The clients are used for interaction between the developers and the Scheduler via event handling mechanisms. The Scheduler may actively send messages to the clients, i.e. the system extends the typical client server architecture towards an agent-oriented structure.

Building the experience base is a future topic of our work and the results are too preliminary to be described here. For example, the arrow labeled “packaging of project experience” is a quite complex operation on all kinds of information (e.g., quality models, measurement data, products, reports) generated throughout the project. We will follow the line of work described in [4]. This article concentrates on the concepts behind the Modeler and the Scheduler.

6 MILOS: A Language for Project Planning

This section explains the main ideas of MILOS on how to represent software process knowledge. The rationales for the introduced constructs were already presented in a variety of publications which describe MVP-S [25,26] and CoMo-Kit [28]. Moreover, a recent paper [40] explains the requirements for the language constructs from a common point of understanding.

Project planning means developing a model of how the project should be performed. At the beginning of a new project, a first step creates an *initial project plan*. This plan contains descriptions of process types, definitions of products to be created and a list of the team members involved in the development process. For large-scale projects, a detailed plan cannot be developed before the enactment starts but planning and execution steps must be alternated (R4): Starting with the initial plan the first processes are enacted. Based on the results, the plan is refined and/or extended (see scenario at the end of Section 1).

To model cooperative development processes, our approach uses four basic notions (fulfilling R1): *Process Types*, *Methods*, *Products* and *Resources*. In the following, these terms are defined as far as it is necessary to understand this article omitting syntactical details of our project planning language. Later on in Section 7, our modeling language MILOS is illustrated by an example which is based on the scenario.

Process Types

A process type describes an activity which must be carried out during process enactment to reach the goals of the project. The description of a process type consists of several parts:

Goal. A (textual, informal) description of the goal of the activity which will be accessed by the process performers during enactment. The goal guides developers by explicitly stating what should be achieved during enactment.

Product Parameters. A product parameter will store input, output and modified products during enactment. Inputs are consumed during process enactment to produce the outputs of the activity. Products which can be changed during enactment are stored in the modified parameter list. In the project plan, we are only able to state which type of information is used or must be produced. For every input the flags[1] mentioned in Table 4 are defined. Output parameters of a process may be optional or required.

Flag Name	Meaning
necessary for planning	The input must be available before the planning of the process starts. Planning here means defining a method or choosing one of the predefined methods.
necessary for execution	The input is not needed for planning but it must be available before the execution starts. Execution here means applying the method.
optional	The input is needed neither for planning nor for execution (but it may be helpful to have).

Table 4: Parameter Flags

Context information. A list of references to information which is not changed by the process enactment (e.g. a file containing the coding standards of the company or a reference to manuals).

Precondition. A formal, boolean condition using process and product attributes which must hold before the process enactment may start. Preconditions are, for example, used to check if the inputs fulfil a given requirement.

Invariant. A formal, boolean condition using process and product attributes which must hold during process enactment. An invariant, for example, may predefine that the enactment of a process must not exceed a certain deadline.

Postcondition. A formal, boolean condition using process and product attributes which must be true after the process enactment has finished. Postconditions are, for example, used to check if the output of a task has a desired quality.

Agent bindings. For every process type, the planner should state criteria which must be fulfilled by agents to be allowed to work on the process during enactment. For example, an agent must have skills in Smalltalk-80 programming and belong to department *ZFE 153*. We distinguish two types of agent bindings: Process performer and process supporter. The performer is responsible for the execution of the process and has to produce the outputs. He may be supported by (several) other agents.

Attributes. An attribute describes a feature of the process type, e.g. the time needed for its enactment.

Methods. A list of alternative courses of action which can be used to reach the goal. The process type describes *what* has to be done, methods describe *how* it can be done.

Products

To model products a standard object-centered approach is used. As usual, we distinguish between types and instances (for sake of brevity, we will use the term “product” for “product instance”).

1.The flags are mutually exclusive.

Product types define a set of slots to structure the product. Every slot is of a particular type. Basic types (e.g., STRING, REAL, ...) are used to specify data. Product types defined by the modeler are used to aggregate products thus forming abstraction hierarchies. During process enactment we represent product instances as values which are assigned to parameters. The type of a parameter is specified by a product class. Product attributes are used to describe qualities of the product, e.g. the complexity of a module.

Methods

A method describes *how* the goal of a process can be achieved. For every process type, the project plan may contain a set of (predefined) alternative methods[1].

Methods are executed by agents (see below). Not every agent who can be responsible for a process may have the abilities to use every method (For example, an implementation process may be enacted by the methods “Implement in C++” or “Implement in Smalltalk”). Therefore, we allow to describe additional agent bindings for every method.

We distinguish between atomic (or elementary) and complex (or composed) methods. The first kind of methods is used to assign products to parameters. *Process scripts* describe how a given task can be solved by a human. *Process programs* are specified in a formal language so that computers can solve a task automatically without human interaction. For an atomic method it is possible to specify what (software engineering) tools are used during enactment. The second kind of methods, complex methods, describe the decomposition of a process into several subprocesses. For every (sub)process type, its cardinality is given which determines how many instances of this process will be created during enactment. Both product types and process types represent building blocks of process models. Mechanisms should be provided in order to link instances together to build a project plan. Therefore complex methods contain two ways of mapping (or binding) products to processes. *Process interface relations* are used to specify the mappings of slots of subprocesses (i.e., output parameters of one subprocess are bound to input parameters of the other subprocesses); a horizontal product flow is specified. *Product mapping* expressions are used to map levels of abstraction. Parameters of subprocesses are mapped onto parameters of the process which contains the complex method. n:1-mappings are allowed which specify aggregation of products. The product mapping must be *level complete,* i.e. each product appears in an expression. In Figure 5 the relation between a process type, one method and the subprocesses is illustrated. Note, that level completeness is achieved because process interface relations allow for transitivity of product mappings (e.g., P-3 is mapped onto P-a by the bindings of P-2). Rules are defined which restrict mappings and allow for consistent bindings (e.g., it is not allowed to map a product consumed by the process to an output parameter of one of its subprocesses).

Finally, a complex method describes how attributes of the subprocesses can be used to compute attributes of the superprocess (attribute mapping).

1.For example, reusable methods may be extracted from old project traces and stored in the experience base. Later on the methods are incorporated into the project plan.

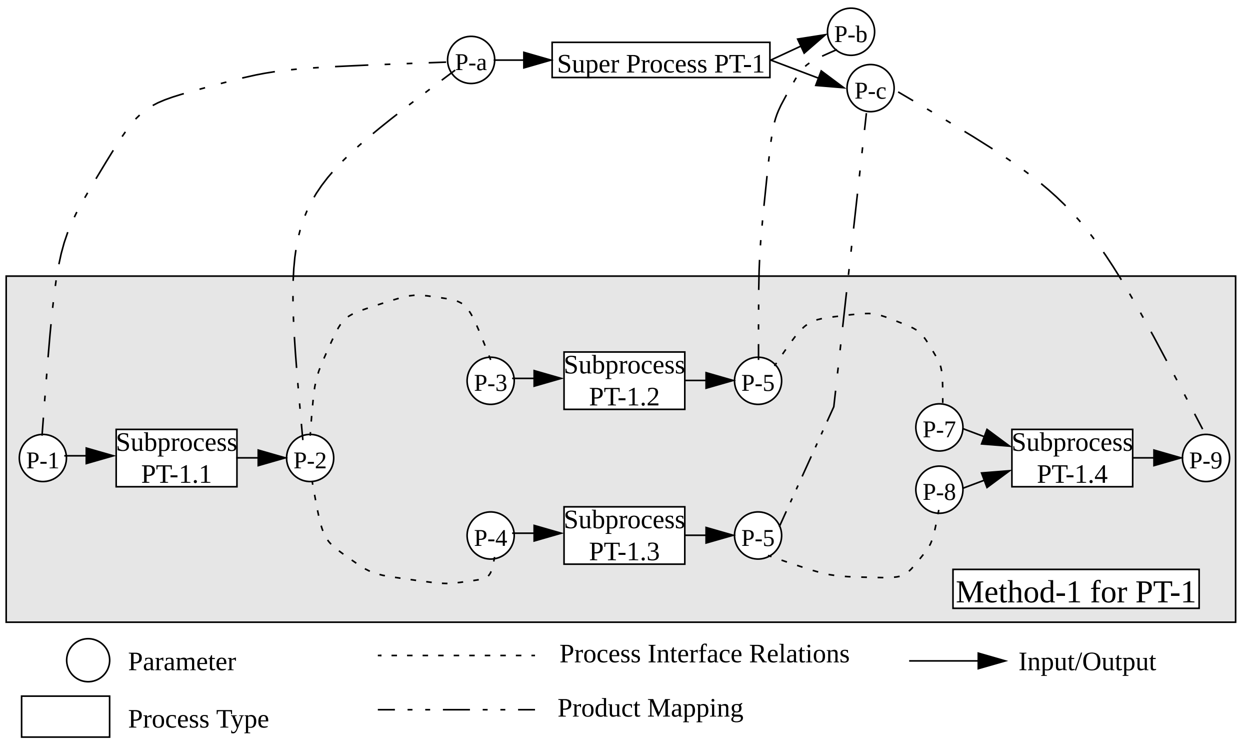


Fig. 5: Vertical and horizontal mapping of parameters

Resources: Agents & Tools

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

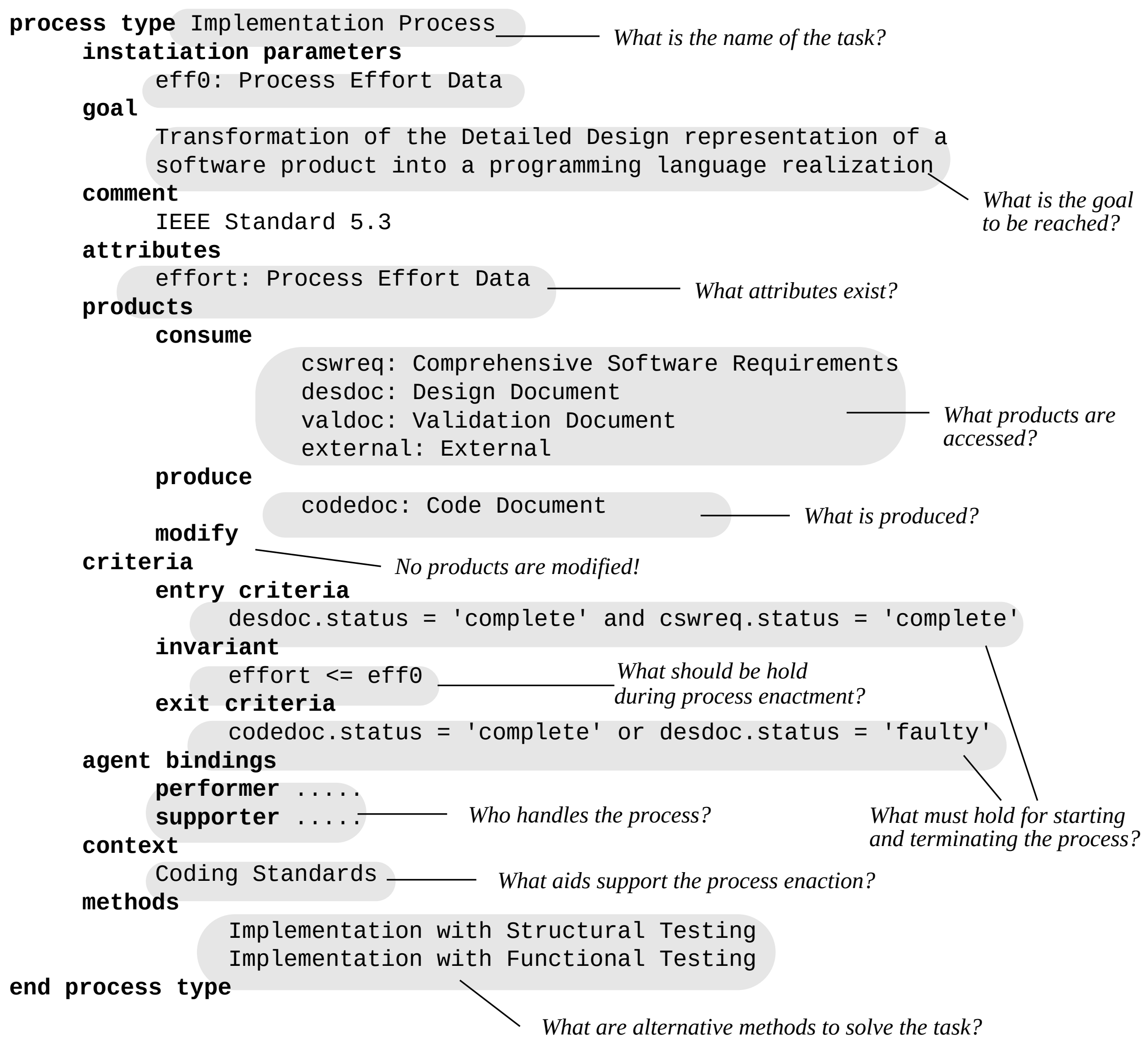
Processes are either performed by *actors* (= human agents) or by *machines*. The first is called „enactment“, the latter „execution“.

For every process type, 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). For example, in a project plan, it may be defined that the process type “implement user interface” should be executed by an actor who has skills in using the Visualworks Interface Builder (q), is a programmer (r), and works in department *ZFE 153* (o). During process execution, our system compares the required properties of a process with the properties an agent possesses. This allows to compute a set of agents who are able to solve the task.

Having sketched our language for project planning (which is basically an extension of MVP-L with methods and object-oriented data modelling facilities), we will now give an example of its use before we explain how the enactment of plans is supported.

7 A MILOS Example

Figure 6 shows the reference process as part of a MILOS adaption of the scenario mentioned in Section 1. The intended representation for modeling in MILOS is graphical. For the purpose of a compact description the example here is given in a textual representation.

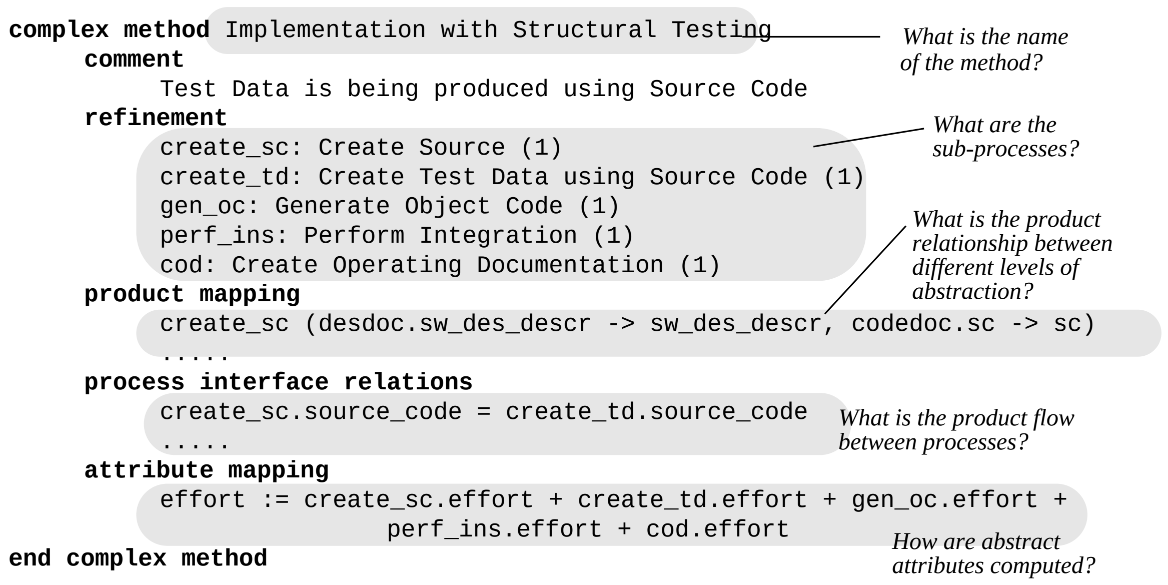


```
process type Implementation Process
     instatiation parameters
          eff0: Process Effort Data
     goal
          Transformation of the Detailed Design representation of a
          software product into a programming language realization
     comment
          IEEE Standard 5.3
     attributes
          effort: Process Effort Data
     products
          consume
                    cswreq: Comprehensive Software Requirements
                    desdoc: Design Document
                    valdoc: Validation Document
                    external: External
          produce
                    codedoc: Code Document
          modify
     criteria
          entry criteria
               desdoc.status = 'complete' and cswreq.status = 'complete'
          invariant
               effort <= eff0
          exit criteria
               codedoc.status = 'complete' or desdoc.status = 'faulty'
     agent bindings
          performer .....
          supporter .....
     context
          Coding Standards
     methods
               Implementation with Structural Testing
               Implementation with Functional Testing
end process type
```

What is the name of the task?
What is the goal to be reached?
What attributes exist?
What products are accessed?
What is produced?
No products are modified!
What should be hold during process enactment?
Who handles the process?
What must hold for starting and terminating the process?
What aids support the process enaction?
What are alternative methods to solve the task?

Fig. 6: Excerpt of a MILOS example: Implementation process with two alternatives

To perform the implementation process two alternative methods are offered. Depending on a decision which can be influenced by measurement data, one of these methods can be applied, e. g., implementation with structural testing (see Fig. 7). This method is complex and it refines the implementation process into several subprocesses.



```
complex method Implementation with Structural Testing
     comment
          Test Data is being produced using Source Code
     refinement
          create_sc: Create Source (1)
          create_td: Create Test Data using Source Code (1)
          gen_oc: Generate Object Code (1)
          perf_ins: Perform Integration (1)
          cod: Create Operating Documentation (1)
     product mapping
          create_sc (desdoc.sw_des_descr -> sw_des_descr, codedoc.sc -> sc)
          .....
     process interface relations
          create_sc.source_code = create_td.source_code
          .....
     attribute mapping
          effort := create_sc.effort + create_td.effort + gen_oc.effort +
                         perf_ins.effort + cod.effort
end complex method
```

What is the name of the method?
What are the sub-processes?
What is the product relationship between different levels of abstraction?
What is the product flow between processes?
How are abstract attributes computed?

Fig. 7: Complex method describing an alternative refinement

8 Supporting Planning and Enactment: The MILOS Scheduler

Based on the requirements in Section 4 we implemented a prototype workflow engine, the MILOS Scheduler. Its main features are:

• It provides the people involved in the project with relevant plan, process and context information to optimize their work and to reduce the information procurement time. This includes the distribution of the processes to the appropriate employees and the presentation of information needed for enactment, e.g. products and process goal descriptions.

• It reduces coordination effort of each team member by notifying the process performer of events (for example plan changes and modified products) he is affected by.

• It allows its users to reject (planning) decisions, and to extend and to modify the initial plan.

• Process knowledge managed by the scheduler allows to guide the project members in their activities.

• It integrates process and measurement technologies.

8.1 Scheduler Support for Different User Groups

The scheduler as a central component of the new process-sensitive software engineering environment supports different roles in a software development project.

- *Project planners* are able to define new processes to be performed to reach the projects' goals. They may access information about the current state of the project, the reasons which led to it, and dependencies between processes. This information is valuable for changing the project plan in case of an undesired situation (e.g. product changes, new requirements from the customer).

- The Scheduler enables *project managers* to delegate processes to their team members and to supervise their enactment. The Scheduler automatically notifies team members of project changes and therefore reduces the manager's coordination effort.

- For *team members* the Scheduler provides a To-Do agenda. When a team member accepts a process waiting for execution, the Scheduler generates a work context which guides him in his activities and allows him to access relevant products and tools. The Scheduler informs him about product modifications which are relevant for his work and notifies him if the process is removed (by the project planner) from the current project plan.

8.2 Operationalizing the Project Plans

The following describes the semantics of the MILOS system in a quite informal manner. This is due to the focus of the article, i.e. explaining the history and integration of the MILOS system. More detailed information about the interpretation of the process models, especially how knowledge-based techniques are employed to support software development, can be found in [13,30]. The most important entities managed by the scheduler are processes, methods, decisions for methods and products. During execution processes and methods passes through various states. Every state is determined by the activities done by the process performers and the dependencies to other processes and decisions. State changes are caused by both activities of process performers and state changes of related entities.

Processes

Precondition. A process is enabled as long as all entry criteria are valid. Only an enabled process can be accepted and enacted.

Delegation. Within a process definition an agent binding is specified. During execution, this binding is evaluated to obtain the set of agents permitted to execute the process. The set is determined by matching the agent bindings with the properties of the agents. The resulting set of authorized agents is further reduced by the project manager. He delegates the process to a subset of potential process performers. Agent bindings and delegations are not static. If an agent binding or delegation changes before the process has been accepted, the set of authorized agents is adapted. If a delegation changes after a performer has accepted the task, the new performer takes over the role of the old one without changing the process state.

Method selection and rejection. In order to reach the process goal, the project planner selects an applicable method. In Section 7 the Implementation Process knows the two methods Implementation with Structural Testing and Implementation with Functional Testing. The set of applicable methods is a subset of the methods defined in the project plan. This set may be reduced depending on the current project context. Selecting a method results in a *decision*. A valid decision is part of the actual project plan. The decision for a method can be rejected later. Such a change activity has consequences on other parts of the project because of dependencies between processes, methods and products. Additionally, the method set can be modified by adding or removing methods from the process specification.

Process invariants. The process invariants have to stay valid during the process performance. If they become invalid, the project planner has to be informed in order to change the current project plan, to assign additional resources, or to initiate other appropriate reactions.

Postcondition. After the work on a process has been finished, its exit criteria are checked. If this checking fails an exception event is forwarded to the process manager who has to resolve this conflict, for example by replanning the process.

Methods

Applying complex methods. A complex method refines a process into a set of one or more subprocesses. Applying a complex method, each subprocess is instantiated *x*-fold. The *x* is determined by the cardinality of the subprocess. In the model, each subprocess has a definite or ∞ cardinality. The cardinality ∞ is replaced during process execution by a definite value. The consumed, produced and modified product parameters of processes with a cardinality greater than one are identified by a definite index.

Applying atomic methods. Atomic methods produce products or data. The name and type of the products that have to be produced are specified in the project plan. If an atomic method is applied, the resulting products are assigned to the corresponding parameter. Additionally, a dependency between the decision for the method and the produced products is established: the rejection of the decision results in the retraction of the parameter assignments.

Product mapping of complex methods. The product mapping of complex methods allows for product exchange between the superprocess and its subprocesses (see also Figure 5). Figure 7 shows that product desdoc.sw_des_descr is mapped to product sw_des_descr. The mapping direction is given by the direction of the arrow sign.

Product interface relations. The product interface relations specify the exchange of products between the subprocesses of a complex method. If a product parameter pair is specified in the interface relations, the product assigned to one of the parameters is automatically assigned to the other one, too.

Agent bindings The agent binding to methods is a subset of those specified in the corresponding process. The qualification of process performers and process supporters expressed by attributes have to fulfill the respective requirements of the methods. This semantic leads to the effect, that an agent who accepts a task may only apply a subset of the specified applicable methods.

8.3 Managing Causal Dependencies

The important relationships MILOS manages explicitly are causal dependencies between processes and subprocesses, and product flow dependencies. MILOS captures these dependencies when agents make decisions (e.g., either functional or structural testing is performed).

Dependencies between Processes and Subprocesses

Complex methods decompose processes into a set of subprocesses. If an agent decides to apply a complex method the subprocesses related to the method become part of the actual project plan. A process that is part of the actual plan is called "valid". The validity of the subprocesses depends on the decision for the corresponding complex method. Therefore we establish a dependency between the validity of the subprocesses and the decision for the corresponding method. If the decision for the method is rejected, the rationale for the validity of the resulting subtasks is no longer given. The subprocesses become invalid. Decisions which have been taken within the subprocesses must be retracted, too.

Product Flow Dependencies

As described above a product assignment is dependent on the decision for the corresponding atomic method. The products produced by an atomic method enter in further decisions and constitute dependencies. The dependencies become important for the process execution, if product producing decisions are rejected. For details see [11,30]. The causal dependencies that are extracted from the process models can be formulated as logical implications. The formalization allows to store the required process information within an „intelligent memory" and to automate the notification of users after changes. Another advantage is that the semantic of the dependencies is made explicit.

The general planning and design model REDUX [30] is the basis for the management of dependencies. REDUX provides mechanisms to decompose processes and to easily change decomposition decisions. Because dependencies between decisions are managed, dependency-directed backtracking is enabled. From CoMo-Kit we have taken over the product flow dependency management [29]. Both approaches use a Justification-based Truth Maintenance System (JTMS [14]) known from Artificial Intelligence to handle the effects of changes efficiently. In a JTMS knowledge about the "world" is represented in a set of beliefs. They are inferred from a set of assumptions that may change over time; that means the world is understood as non-monotonic, i.e. changing. Assumptions are axioms (or facts) describing basic elements of the world. For each belief a justification explains how it was interfered. The justifications therefore record dependencies among the beliefs. If any assumption changes the JTMS updates the beliefs. Only beliefs that are affected from a change are examined. The set of beliefs to update is determined by the justifications (i.e., dependencies). MILOS uses the REDUX approach and components of CoMo-Kit to infer the status (e.g., validity) of processes, product assignments, decisions, and all other MILOS concepts. The causal dependencies between the objects are represented as justifications. Because changes in MILOS project plans usually have a limited scope of effect, the state of objects affected from a change is updated efficiently. In [30] a detailed description of the mechanisms is provided.

8.4 Flexible Planning

Supporting flexible planning is a key feature of the MILOS project. MILOS allows to refine and change the initial plan during enactment. Coordinating and managing causal dependencies is a prerequisite to support plan changes. In this section, a few examples for dynamic planning with MILOS are given. For a detailed description see [12, 13].

Refining the Plan

One goal is to allow the planner and the project manager to delay planning decisions to the execution phase, when project specific information is available. Within MILOS there are the following possibilities to do so: By defining more than one method to solve a goal or by extending the initial model during enactment.

- *making decisions:* In the example of Figure 6, two alternative methods Implementation with Structural Testing and Implementation with Functional Testing are defined for the process Implementation Process. The decision for one of the alternatives is made during execution. The planner can use current process knowledge for decision support.

- *extending the model.* During execution, increasing process knowledge may result in new solutions. In MILOS, the planner can add new methods to the process model or refine existing ones while the model is enacted. The new method is inserted into the dependency network of the scheduler and immediately available to the current project. In the example shown in Figure 6, the planner might add a new method named Implementation With Equivalence Class Testing for the Implementation Process and select the method afterwards.

Changing the Plan

During project enactment changing conditions and planning errors lead to discarded solutions. These changes affect the project plan as well as the produced products. As a result, the plan has to be adapted to the new situation in order to represent the project accurately. At this point, the benefit of managing dependencies between events and plan states becomes visible: the effects of plan changes are handled by the system, affected team members are notified, and require guidance on how to react in an appropriate way. Our approach provides several alternatives of changing the plan: rejecting and making decisions or changing the process model.

- *changing decisions:* Imagine, that in the above example the project planner already applied the method Implementation with Structural Testing (see Figure 6) after analyzing the consumed product desdoc. After selecting the method, he recognizes, that this method would exceed the effort limit specified within the invariant. Therefore, he rejects the decision and selects the other method. Because the Scheduler tracks product flow dependencies and dependencies between decisions and subprocesses (see above), the effects of the changes are propagated through the dependency network. For example, the subprocesses create_sc, create_td, gen_oc, per_ins, cod of method Implementation with Structural Testing (Figure 7) change their state: they are not longer part of the actual project plan. The team members who have been working on these processes are notified of the state change.

- *changing the model.* There are cases in which changing decisions as described before is insufficient: modeling errors or incomplete models. Often these kinds of errors are noticed after starting the enaction of the plan. Nevertheless the errors have to be corrected to guarantee an accurate enactment which means the project state representation has to be a model of the real-world project. Such errors can be eliminated by changing the process models. MILOS allows to modify method definitions, process types, and agent bindings. The current implementation allows to correct errors within method or process definitions only if the affected parts of the entities are not executed yet. In the example of Figure 7 the planner could remove process create_td if it was not already enacted. Currently we extend MILOS to tackle consequences of changes of already enacted processes (i.e., problems P1 and P3, see Section 4, are not addressed for the reasons explained above).

8.5 Discussion of the Requirements

MILOS is based on the MVP-L approach. Therefore, the main concepts of software development processes are supported (R1). The MILOS Scheduler supports project planners and managers (R2, R3). Extending, modifying, and changing the current project plan is supported by the MILOS Scheduler using CoMo-Kit techniques. Planning and enactment decisions are handled by the Scheduler (R5). Because the dependencies point to the cause of an event, the Scheduler is able to guide the user in appropriately reacting to changes (R6). Alternating planning and enactment steps is supported (R4). MILOS provides an object-centred product model and the Scheduler is able to store and manage products (R7). Process and product attributes and the attribute mappings support measurement activities (R8). By using a workflow engine, the Scheduler, team members are guided and coordinated in their activities. Activities for reacting to events may itself have global effects on the project planning and execution. Because the Scheduler manages dependencies, the performer is relieved of coordination activities resulting from such changes (R9, 10). The MILOS language has been designed to automate process steps by using process programs (R11). Currently, this feature is not yet implemented in the Scheduler, but it will be a future extension.

9 Related Work

To check for completeness of the MILOS language we compared it with existing frameworks and definitions that were developed by Conradi, Fernström and Fuggetta [9], Feiler and Humphrey [15], Lonchamp [23], and Armitage and Kellner [2] (see Table A.1 of the appendix). Every framework provides a consistent set of concepts that embodies a particular understanding about aspects of software processes. In contrast to the concepts presented in this article, aspects of the meta-process (i.e., the process of process modeling) are also described in some of the papers [9, 15, 23]. The four definition frameworks are not formalized but natural language is used to explain the meaning of their concepts. Because the terms were developed in different contexts, one cannot assume a perfect match between them. Therefore, we see the terms from different frameworks as similar, not as equal. Under this assumption, MILOS implements most of the concepts covered by the other frameworks. No predefined type classifying all components of the delivered product (i.e., as proposed as Deliverable by Lonchamp) is present in any approach. All other abstract concepts covered by the frameworks are considered in MILOS.

Process-sensitive software engineering environments which support evolution of enacted process models are a focal point of current research, but the results are still immature [27, 37]. In the remainder of this section, we discuss environments relevant for our work, and point out the main differences to our approach. Important requirements not met by the related approaches are checked (which leaves open whether the other requirements are met). The unsatisfied requirements are marked by a '¬'.

The SPADE environment is a system for developing analyzing, and enacting process models described in the language SLANG (Spade LANGuage) [3]. *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. Whenever the type of an active process is modified, SPADE prompts the user to provide a transformation function. This is a solution for the problem P2 (i.e., type-state correspondence) which is not solved by our approach. SLANG provides only a small set of software development process concepts (¬R1). Also the user must decide when to start process evolution. The system does not provide any support to decide which parts need to be changed (¬R5).

GRAPPLE is an operator-based approach which supports planning and plan recognition [19]. The operators encapsulate the functionality of both tools and processes performed by agents. Reason maintenance techniques are used to manage dependencies between process steps; dependencies between products are not maintained (¬R5). Our synthesized approach extends these techniques by explicitly representing dependencies between products, so that a goal-directed reaction to changes of the product state is possible. GRAPPLE does not support the alteration of planning and enactment (¬R4).

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 [21]. 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. Thus the EPOS Process Modeling System tackles the problems P1 and P2. No dependencies between process fragments are managed (¬R5), so that it is not possible to determine what processes accessed a faulty product and might have to be enacted another time. Detection of deviations and recognition of a change's impact are completely left to the user (¬R6).

Redoing 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 [38]. 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 enacted instead and anew. The decision to redo is specified in the process models. It can be seen as a sort of "goto" where results in the subtree are discarded. In contrast to our approach, short-term planning is not supported (¬R3), the process models must be completely defined before interpretation (¬R4), and the decisions for redoing are predefined, which means that criteria to detect deviations from the plan must be specified within the models (¬R5).

10 Summary and Future Work

This article presents the integration of two process support approaches, namely CoMo-Kit and MVP-E. They both were developed independently, to solve particular and isolated problems of automated process support. A recent comparison revealed commonalities and differences of both systems [40]. Requirements were set up for process-sensitive software engineering environments. They are addressed by the newly created approach MILOS which is a synthesis of CoMo-Kit and MVP-E. The concepts of MILOS were illustrated, using a scenario of a standard implementation process. By relating MILOS to other approaches the uniqueness of this approach was shown. On the other hand problems were encountered and identified as potential future work within our research work.

Roughly spoken, as an intermediate result of the synthesis MVP-L’s concepts are used to describe software processes and the concepts of the CoMo-Kit process engine are used to enact the models. Using these basic elements, we are able to alternate modeling, planning, and enaction modes. Our integrated approach provides a sufficient set of concepts to capture real-world processes. By integrating knowledge based techniques a flexible process-sensitive software engineering environment will be created which manages dependencies between project information and supports backtracking.

The MILOS system is still in an immature state, for example it lacks functionality which a process-sensitive software engineering environment should have and which are proposed in this article (i.e., cf. R11 and P1 to P3). Moreover, the MILOS system was not yet validated in a real software development project. Although some experiments and case studies have been performed in both our work groups to evaluate the usefulness of both MVP-S and CoMo-Kit (see for example [25] for an analysis of MVP-S and [31] for a case study of CoMo-Kit), an in-depth analysis whether the MILOS system fulfills the expectations is still needed. In fact, the validation of the system is recognized as an important goal and will be performed in the near future. The development of a family of control software systems for building automation will serve as a test environment for the validation of the MILOS system. The future experiments have the purpose to evaluate the applicability of the ideas presented in this article and to identify potential future research in order to improve the designed process-sensitive software engineering environment.

Acknowledgments: Sigrid Goldmann did careful work comparing CoMo-Kit and MVP-L. She also helped to improve the article’s readability. The anonymous reviewers gave important comments to improve the article. The authors would like to thank the members of both work groups who contributed to many ideas presented in this article.

References

[1] P. Armenise, S. Bandinelli, C. Ghezzi, and A. Morzenti. Software process languages: Survey and assessment. In *Proceedings of the Fourth Conference on Software Engineering and Knowledge Engineering*, Capri, Italy, June 1992.

[2] James W. Armitage and Marc I. Kellner. A conceptual schema for process definitions and models. In Dewayne E. Perry, editor, *Proceedings of the Third International Conference on the Software Process*, pages 153–165. IEEE Computer Society Press, October 1994.

[3] 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.

[4] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. Experience Factory. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, pages 469–476. John Wiley & Sons, 1994.

[5] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. Goal Question Metric Paradigm. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, pages 528–532. John Wiley & Sons, 1994.

[6] Victor R. Basili and H. Dieter Rombach. The TAME Project: Towards improvement–oriented software environments. *IEEE Transactions on Software Engineering*, SE-14(6):758–773, June 1988.

[7] Israel Z. Ben-Shaul, Gail E. Kaiser, and George T. Heineman. An architecture for multi-user software development environments. In H. Weber, editor, *Proceedings of the Fifth ACM SIGSOFT/SIGPLAN Symposium on Software Development Environments*, pages 149–158, 1992. Appeared as ACM SIGSOFT Software Engineering Notes 17(5), December 1992.

[8] Alfred Bröckers, Christopher M. Lott, H. Dieter Rombach, and Martin Verlage. MVP–L language report version 2. Technical Report 265/95, Department of Computer Science, University of Kaisers-lautern, 67653 Kaisers-lautern, Germany, 1995.

[9] Reidar Conradi, Christer Fernström, and Alfonso Fuggetta. A conceptual framework for evolving software processes. *ACM SIGSOFT Software Engineering Notes*, 18(4):26–35, October 1993.

[10] Bill Curtis, Marc I. Kellner, and Jim Over. Process modeling. *Communications of the ACM*, 35(9):75–90, September 1992.

[11] Barbara Dellen, Kirstin Kohler, and Frank Maurer. Integrating software process models and design rationales. In *Proceedings of the Knowledge Based Software Engineering Conference*, pages 84–93, 1996.

[12] Barbara Dellen and Frank Maurer. Integrating planning and execution in software development processes. In *Proceedings of the Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE '96)*, pages 170–176. IEEE CS Press, June 1996.

[13] Barbara Dellen, Frank Maurer, and Gerd Pews. Knowledge based techniques to increase the flexibility of workflow management. *Special Issue of the Data & Knowledge Egnieering Journal*, 1996. to appear.

[14] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[15] Peter H. Feiler and Watts S. Humphrey. Software process development and enactment: Concepts and definitions. In *Proceedings of the Second International Conference on the Software Process*, pages 28–40. IEEE Computer Society Press, February 1993.

[16] Christer Fernström. Process WEAVER: Adding process support to UNIX. In *Proceedings of the Second Interna-*

tional Conference on the Software Process, pages 12–26. IEEE Computer Society Press, February 1993.

[17] Pankaj K. Garg and Mehdi Jazayeri. *Process-centered Software Engineering Environments*. IEEE Computer Society Press, 1996.

[18] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed & Parallel Databases*, 3:119–153, 1995. Kluwer Academic Press, Boston.

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

[20] Institute of Electrical and Electronics Engineers. *IEEE Standard for Developing Software Life Cycle Processes*, 1992. IEEE Std. 1074-1991.

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

[22] C. D. Klingler, M. Neviaser, A. Marmor-Squires, C. M. Lott, and H. D. Rombach. A case study in process representation using MVP–L. In *Proceedings of the Seventh Annual Conference on Computer Assurance (COMPASS 92)*, pages 137–146, June 1992.

[23] Jaques Lonchamp. A structured conceptual and terminological framework for software process engineering. In *Proceedings of the Second International Conference on the Software Process*, pages 41–53. IEEE Computer Society Press, February 1993.

[24] Christopher M. Lott. Measurement support in software engineering environments. *International Journal of Software Engineering & Knowledge Engineering*, 4(3):409–426, September 1994.

[25] Christopher M. Lott. *Measurement-based feedback in a process-centered software engineering environment*. PhD thesis, Department of Computer Science, The University of Maryland, College Park, Maryland 20742, February 1996.

[26] Christopher M. Lott, Barbara Hoisl, and H. Dieter Rombach. The use of roles and measurement to enact project plans in MVP-S. In W. Schäfer, editor, *Proceedings of the Fourth European Workshop on Software Process Technology*, pages 30–48, Noordwijkerhout, The Netherlands, April 1995. Lecture Notes in Computer Science Nr. 913, Springer–Verlag.

[27] Nazim H. Madhavji and Maria H. Penedo. Guest editor's introduction. *IEEE Transactions on Software Engineering*, 19(12):1125–1127, December 1993. Special Section on the Evolution of Software Processes.

[28] Frank Maurer. *Hypermedia-based Knowledge Engineering for distributed, knowledge-based Systems*. PhD thesis, Universität Kaiserslautern, 1993. In German.

[29] Frank Maurer. Project coordination in design processes. In *Proceedings of the Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE '96)*, pages 191–196. IEEE CS Press, June 1996.

[30] Frank Maurer and Jürgen Paulokat. Operationalizing conceptual models based on a model of dependencies. In A. Cohn, editor, *Proceedings of the 11th European Conference on Artificial Intelligence*. John Wiley & Sons, Ltd., 1994.

[31] Frank Maurer and Gerhard Pews. Supporting cooperative work in urban land–use planning. In *Proceedings of COOP–96*, 1996.

[32] Andreas Oberweis. Workflow management in software engineering projects. In S. Medhat, editor, *Proceedings of the 2nd International Conference on Concurrent Engineering and Electronic Design Automation*, 1994.

[33] Ch. Petrie. *Planning and Replanning with Reason Maintenance*. PhD thesis, University of Texas, Austin, 1991.

[34] Charles Petrie. Context maintenance. In *Proceedings of the AAAI–91*, 1991.

[35] H. Dieter Rombach and Martin Verlage. How to assess a software process modeling formalism from a project member's point of view. In *Proceedings of the Second International Conference on the Software Process*, pages 147–158, February 1993.

[36] H. Dieter Rombach and Martin Verlage. Directions in software process research. In Marvin V. Zelkowitz, editor, *Advances in Computers, vol. 41*, pages 1–63. Academic Press, 1995.

[37] Stanley M. Sutton, Jr., Dennis Heimbigner, and Leon J. Osterweil. Language constructs for managing change in process–centered environments. In *Proceedings of the Fourth ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 206–217, 1990. Appeared as ACM SIGSOFT Software Engineering Notes 15(6), December 1990.

[38] 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.

[39] Martin Verlage. Multi–view modeling of software processes. In Brian C. Warboys, editor, *Proceedings of the Third European Workshop on Software Process Technology*, pages 123–127, Grenoble, France, 1994. Nr. 772, Springer–Verlag.

[40] Martin Verlage, Barbara Dellen, Frank Maurer, and Jürgen Münch. A synthesis of two process support approaches. In *Proceedings of the 8th Software Engineering and Knowledge Engineering Conference (SEKE'96)*, pages 59–68. Knowledge Systems Institute, Skokie (IL), USA, June 1996.

Appendix

Table A.1 relates software process terms defined by different authors. The terms explain real-world concepts. The definitions given in [2, 9, 15, 23] are in natural language and therefore lack formality. The overview presented in Table A.1 should not be understood as a precise comparison of terminology. The terms differ slightly in their meanings even when they have the same name. The matching was performed based on careful but subjective assessment. The reader is referred to the cited literature for a detailed explanation of the terms. A table cell two high means that the term corresponds to two terms of another framework. Empty cells mean that no term with an equivalent meaning to other terms of that row is defined in the approach discussed in the column.

MILOS	CoMo-Kit	MVP-L	Armitage and Kellner [2]	Feiler, Humphrey [15][a]	Lonchamp [23][a]	Conradi et al. [9][a]
Process	Task	Process	Process (instance)	Process (Element)	Software Process	Process
Process Type		Process Model	Process (type)	Process Definition		Template
						Production Process
Method	Method		Activity		Process Step	Activity
			Activity Description	Process Script		
			Procedure	Process Program		
Atomic Method	Atomic Method	Elementary Process[b]		Process Step	Activity	
				Task	Task	
Process Attribute		Process Attribute	Activity State[c]			
Complex Method	Complex Method	Refinement	Decomposition			
Precondition Invariant Postcondition		Criteria	Behavioral Information	Process Constraint	Constraint	
Project Plan		Project Plan		Process Plan		
				Project Plan		Software Project
Product (Instance)	Concept Instance	Product	Artifact (instance)		Artifact	Artifact (Input)
						Software Item (Output)
Product Type	Concept Class	Product Model	Artifact (type)			Template
Product Slot	Attribute	Elementary Product				
						Software Product
					Deliverable	
Product Attribute		Product Attribute	Artifact Statec			
Product Flow	Information flow	Product Flow	Artifact Flow			
Resource		Resource			Resource	
Agent	Agent	Personnel	Agent	Agent	Agent	Agent
Tool		Tool				Tool
Property		Resource Attribute	Agent Statec			
Attribute		Attribute	Attribute[d]			
		Model	Process Definition	Process Definition	Generic Process Model	Template

Table A.1: Relating Different Frameworks

a. Not all process terms presented in [9, 15, 23] are considered in the corresponding columns, because many of them describe no concept of software development processes but ideas of enaction (e.g., enactment state), organizational processes (e.g., monitoring), or process characteristics (e.g., liveness).

b. Process which contains no refinement.

c. Predefined attributes used by a process engine (interpretation machine) to manage an overall project state.

d. Attribute is explained as “a textual description of information”. This general and abstract definition matches the other terms in the row only partially.

The Sonderforschungsbereich 501

The “Deutsche Forschungsgemeinschaft (DFG)” is a major sponsor of basic research activities in Germany. Besides individual projects DFG sponsors long-term strategic research activities at German universities. The most prestigious form of funding are so-called “Sonderforschungsbereiche (SFBs)”, special research institutes aimed at addressing fundamental research areas. Specific characteristics of SFBs include their affiliation with a highly respected scientific department at a German university, funding periods of 9 to 15 years (with regular evaluations), and interdisciplinary collaboration.

The SFB 501 on “Development of Large Systems with Generic Methods” was started at the University of Kaiserslautern on January 1, 1995. It aims at developing and evaluating a set of techniques, methods and tools for supporting the fast and reliable customization of complex domain specific software systems. The emphasis is on techniques, methods and tools that support reuse of all kinds of software artifacts ranging from system components to process fragments and other related knowledge.

In the first step existing techniques, methods and tools are being evaluated for suitability within the domain of process control - starting with the application scope of building automation.

The mid-term goal of the SFB is to generalize the resulting techniques, methods and tools such that they can be used within other application scopes and domains to establish similar reuse-based development processes.

The long-term goal of the SFB is to contribute to the science base for transforming software development from an art to an engineering discipline.

Within the SFB several research groups from the departments of Computer Science and Electrical Engineering collaborate. Current employment count includes 8 professors, 17 full-time researchers and about the same number of part-time student assistants. The following projects have been established:

- Application scope “Building Automation”: A project to investigate requirements on such systems and build simulation models for system/acceptance testing
- Software Engineering Laboratory: A project to support both the prototype development of a first building automation system and conducting experiments
- Experiment-based modeling of software development processes/knowledge based planning and control of SE processes: Two projects to support the planning and execution of processes for software development and experimentation based on existing knowledge
- Generic communication systems/Generic system software: Two projects to investigate the possibilities for generic modeling of system software needed within the chosen application domain
- Formal description techniques: A project to select, modify, integrate and evaluate the appropriate description techniques for generic development

Additional projects will be proposed at the next SFB review. The work discussed in this article is a very first result of a collaboration of the second and third projects listed above.