A Generalized Workflow System for Mechanical Engineering*

Peter Heimann and Bernhard Westfechtel
Lehrstuhl für Informatik III
RWTH Aachen, D-52056 Aachen

E-Mail: [peter|bernhard]@i3.informatik.rwth-aachen.de

Abstract

We present a workflow system which supports development processes in mechanical engineering. While conventional workflow systems have been applied successfully to routine processes in office applications, our system is designed to support creative development processes. In particular, the activities of a workflow are arranged in a task net which depends on the product structure developed so far in the project. Furthermore, feedbacks to earlier steps of the workflow are taken into account (without the need to plan them in advance). Finally, the steps of a workflow may be executed in parallel to shorten development cycles (simultaneous engineering).

Keywords workflow system, process management, engineering data management

1 Introduction

A *workflow system* [4, 6] has been defined as a proactive computer system which manages the flow of work among participants, according to a defined procedure consisting of a number of tasks. To this end, the workflow system keeps track of the current execution state, builds up agendas and provides participants with both data and tools. In this way, the activities of participants are co-ordinated to achieve defined objectives by set deadlines.

Workflow systems have been applied to many different domains, including office automation, computer-aided design, software engineering, etc. In particular, they have been successful in supporting *routine work* which follows highly structured, predefined procedures and is amenable to automation. However, support of *creative processes* is still a challenge. In particular, this holds for development processes in engineering disciplines.

There are different views concerning the relations between the workflow and its participants [9]. Traditionally, participants *execute* tasks defined in the workflow. Recently, it has been recognized that participants also must be able to *define* and modify workflows. In other words, the workflow is also a *product* on which (some) participants are working.

In particular, in development processes the workflow cannot be defined once and for all before development starts. Rather, it evolves according to a variety of factors, including the product structure, feedbacks during execution, changed deadlines, etc. Thus, workflows are highly *dynamic* in that they are not only executed, but are also modified structurally during execution [5].

In this paper, we present a workflow system which supports development processes in mechanical engineering [10, 11]. The workflow system has been developed in the *SUKITS project* [3] carried out by computer scientists and mechanical engineers at RWTH Aachen. It has been designed specifically to support dynamic development processes, and covers management of products and resources as well.

2 Product Management

Product management deals with *documents*, which are artifacts created and used during the development process (designs, manufacturing plans, NC programs, part lists, simulation results, etc.). A document is a logical unit of reasonable size typically manipulated by a single engineer. According to the constraints of a posteriori integration, we do not make any assumptions regarding the internal structure of documents. This approach is called *coarse-grained* because a document is considered an atomic unit.

Related documents (e.g., all documents describing a single part) are aggregated into *configurations*. In addition to components (either documents or subconfigurations), configurations contain *dependencies* as well. Such dependencies may either connect components belonging to the same working area (e.g., dependencies between designs of components of an assembly part), or they may cross working area boundaries (e.g., dependencies between designs and manufacturing plans). Product management records these dependencies so that consistency between interdependent documents can be controlled.

During its evolution history, documents evolve into multiple *versions*. Since documents and configurations are handled uniformly, configurations are versioned too. Versions may be regarded as snapshots

*This work was partially supported by the German Research Council (DFG).

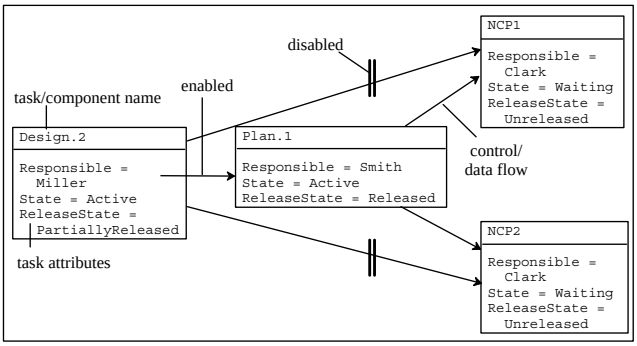

Figure 1: Example of a task net

recorded at appropriate points in time. The reasons for managing multiple versions of an object (instead of just its current state) are manifold: reuse of versions, maintenance of old versions having already been delivered to customers, storage of back-up versions, or support of change management.

Versions of documents or configurations are arranged in *version graphs* whose nodes and edges correspond to versions and successor relationships, respectively. In simple cases, versions are arranged in a sequence reflecting the order in which they were created. Concurrent development of multiple versions causes branches in the evolution history (version tree). Merging of changes performed on different branches results in directed acyclic graphs (dags), where a version may have multiple predecessors. Finally, a version graph may even be separated if multiple branches have been developed in parallel from the very beginning.

3 Process Management

We follow a *product-centered* approach to *process management*. The basic idea is that the product structure also determines the development process. Thus, a configuration version, which consists of interdependent component versions, is enriched with process data, resulting in a *task net*. For each version component, there is a corresponding task which has to produce this component. Dependencies between version components correspond to horizontal task relations which represent both *data* and *control flow*. Thus, a task has inputs defined by its incoming data flows, and it produces a single output, namely the corresponding version component. If a flow relation starts at t_1 and ends at t_2, t_1 and t_2 are denoted as predecessor and successor task, respectively. Finally, tasks can also be connected by (vertical) *composition relations*: version components referring to configuration versions result in hierarchies of task nets. The leaves of the composition hierarchy are called atomic tasks; otherwise, a task is complex.

An example of a task net is given in Figure 1. The task net corresponds to a configuration version which contains one design, one manufacturing plan, and two NC programs. Each task is named by the corresponding version component, i.e., we refrain from introducing explicit task names such `CreateDesign` or `CreatePlan`. Tasks are decorated with attributes, e.g., for representing task states, release states of outputs, and employees responsible for task execution. The design task and the planning task have already produced outputs (bound version components), while both NC programming tasks are still waiting for their execution (unbound version components).

Product evolution results in incremental extensions and structural changes of task nets. For example, when starting development of a single part, it may only be known that one design and one manufacturing plan have to be created. Then, the initial task net only contains these tasks, which are known a priori. Which NC programs must be written, is determined only when the manufacturing plan has been worked out.

Simultaneous engineering [1] is supported by *prereleases* of intermediate results. Since a task may release preliminary versions of its output, successors may start execution when their predecessors are still active. Prereleases serve two purposes. First, the overall development process may be finished earlier because of overlapping task executions. In particular, work located on the critical path may be accelerated. Second, wrong decisions can be detected much earlier by taking feedbacks from later phases in the lifecycle into account (e.g., design for manufacturing).

Versions may be prereleased *selectively* to successor tasks. An example is given in Figure 1, where the design task has the release state `PartiallyReleased`. Attributes attached to flow relations control which successors may access a selectively released version. In Figure 1, double vertical bars indicate *disabled* flows along which outputs may not be propagated. NC programming tasks may not yet access the design because the geometry still has to be elaborated in detail. On the other hand, the flow to the planning task is *enabled* because the manufacturing plan can already be divided into manufacturing steps based on a draft design.

Feedbacks in the development process are taken care of as well. Feedbacks occur when problems and errors are detected which require changes to the outputs of predecessor tasks. For example, the manufacturing planner may come to the conclusion that the part to be produced has not been designed for manufacturing.

In general, it is by no means obvious how to handle feedbacks. A feedback is raised by some task t_1 which detects some problem regarding its inputs. This problem may be caused by some task t_2 which is a transitive predecessor of t_1. To handle the feedback, t_2 must be determined at first. Then, it has to be decided which tasks will be affected by the feedback in which ways. Affected tasks may be suspended or continued without disruption, depending on how radically their inputs are expected to change. Finally, the wave of changes has

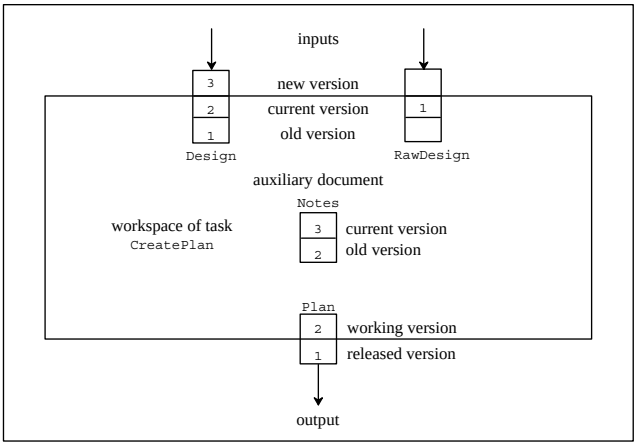

Figure 2: Workspace of a task

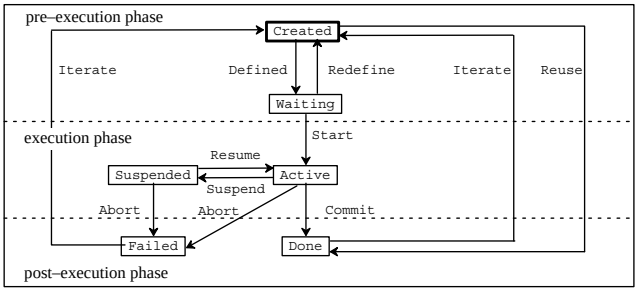

Figure 3: State transition diagram for tasks

to be propagated through the task net.

Feedback handling requires decisions to be performed by managers and engineers. Furthermore, since the current state of the development process needs to be analyzed and changed, feedback handling is an excellent example why the development process has to be made available as a product of management activities.

Due to product evolution, simultaneous engineering, and feedbacks, the *workspace* of a task is highly dynamic. The workspace consists of input documents, the output document, and potentially further auxiliary documents which are only visible locally. All components of the workspace are subject to version control. Multiple versions of inputs may be consumed sequentially via incoming data flows. Similarly, multiple versions of the output document may be produced one after the other. Finally, versions of auxiliary documents may be maintained as well.

The workspace of a sample task is illustrated in Figure 2. For inputs, the *current version* denotes the version which is currently being used by the responsible engineer. A *new version* may have been released already by the predecessor task (see input `Design`). When it is consumed, the current version is saved as an *old version*, and the new version replaces the current one. For outputs, we distinguish between a *working version*, which is only locally visible, and a *released version*, which is available to successor tasks. When the working version is released, it is frozen and replaces the previously released version. Any subsequent update to the working version triggers creation of a successor version. Finally, a current and an old version are maintained for auxiliary documents.

Imports of inputs and exports of outputs are controlled explicitly by the responsible engineer who invokes operations for *consuming* inputs and *producing* outputs. Since only stable versions may be released, the import part of a workspace remains stable as long as no `Consume` operation is executed. In this way, the engineer may control the time of upgrading to a new input version (and may still access the previously current version as an old version, e.g., to perform a `diff` operation in order to analyze the differences). Similarly, changes to outputs are visible only when the working version is released. Thus, versions are used to achieve loose coupling between activities of multiple engineers.

4 Resource Management

Human resources are *employees* who are organized into project teams. Employees are assigned to tasks according to the *roles* they play in a project. Each role (e.g., `NC programmer`) corresponds to a certain task type (e.g., creation of an NC program). In general, each team member may play any set of roles. A task may be assigned to an employee only if (s)he plays the appropriate role.

There is no explicit distinction between *engineers* and *managers*. Typically, engineers and managers are assigned to atomic and complex tasks, respectively. An atomic task corresponds to a *technical task* such as creation of a design or of an NC program. A complex task such as development of a single part is viewed as a *management task*. The manager who is assigned to a supertask is in charge of planning and controlling execution of its subtasks. In particular, the manager has to create subtasks and assign them to engineers of the project team (or to mangers of complex subtasks).

Employees are supported by *technical resources* (tools) in executing assigned tasks. In the case of atomic tasks, these are the development tools tools integrated with the workflow system. In the case of complex tasks, tools of the workflow system itself are used for task execution. Unlike human resources, tools are implicitly assigned to tasks. Each tool is characterized by the types of objects on which it may operate. It is called through an envelope realized as a Perl script.

5 Formal Specification

The workflow system has to perform sophisticated analyses and modifications of complex data structures. Instead of encoding these ad hoc in some programming language, we have prepared a formal specification based on *graph rewriting systems*. To this end, we have used the specification language PROGRES

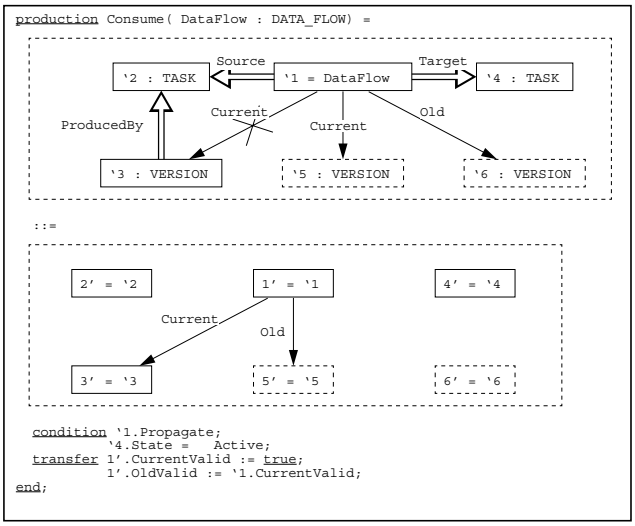

Figure 4: Consumption of an input version

[8] which supports high-level descriptions of complex transformations on typed, attributed graphs. In particular, task nets are represented as graphs, and editing, analysis, and execution are described in a uniform formalism. This is essential to support fine-grained interleaving of these operations, and it distinguishes our approaches e.g. from Petri nets which do not provide any formalism for specifying edit operations [2].

However, in addition to graph rewriting we employ traditional formalisms like e.g. *state transition diagrams* as well. For example, we have developed a state transition diagram for tasks which is specifically designed for managing development processes (Figure 3). `Created` serves as initial state which is also restored in case of feedbacks. The `Defined` transition indicates that the task definition has been completed. In case of feedbacks, `Reuse` skips task execution if the task is not affected. In state `Waiting`, the task waits for its activation condition to hold. `Redefine` returns to `Created`, while `Start` is used to begin execution. Execution of an `Active` task may be suspended, e.g., because of erroneous inputs, and resumed later on. `Failed` and `Done` indicate failing and successful termination, respectively. From both states, re-execution may be initiated by `Iterate` transitions.

An example of a *graph rewrite rule* is given in Figure 4. Application of `Consume` serves to update the workspace of some task by "importing" a new version of some input document. `Consume` is applied to an incoming data flow (node `'1`) with enabled propagation (condition part). It replaces the current version (node `'5`) with the new version (node `'3`) and the old version (node `'6`) with the previously current version (node `'5`). The negative edge between nodes `'1` and `'3` ensures that the new version is not current yet. Nodes `'5` and `'6` are optional. In particular, they are absent when data are propagated along the flow for the first time.

6 Conclusion

The workflow system described in this paper has been fully implemented within the SUKITS project, partially relying on technology developed in the IPSEN project on integrated software development environments [7]. The *front end* supports engineers by displaying agendas, providing workspaces for executing tasks, offering commands for calling external tools, etc. The *management environment* offers graphical views on version graphs and configurations/task nets and provides commands for planning, analysis, and controlling development processes. Finally, the *parameterization environment* serves to adapt the workflow system to a certain application domain, e.g., by defining types of documents and relationships. The implementation of the workflow system comprises more than 60,000 lines of code in Modula-2 and C (in addition to code reused from the IPSEN project).

References

[1] H.-J. Bullinger and J. Warschat, editors. *Concurrent Simultaneous Engineering Systems*. Springer, 1996.

[2] W. Deiters and V. Gruhn. The FUNSOFT net approach to software process management. *International Journal of Software Engineering and Knowledge Engineering*, 4(2):229–256, 1994.

[3] W. Eversheim, M. Weck, W. Michaeli, M. Nagl, and O. Spaniol. The SUKITS project: An approach to a posteriori integration of CIM components. In *Proceedings GI Jahrestagung*, Informatik aktuell, pages 494–503. Springer Verlag, 1992.

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

[5] P. Heimann, C.-A. Krapp, M. Nagl, and B. Westfechtel. An adaptable and reactive project management environment. In Nagl [7], pages 504–534.

[6] S. Jablonski and C. Bußler. *Workflow Management — Modeling Concepts and Architecture*. International Thomson Publishing, Bonn, 1996.

[7] M. Nagl, editor. *Building Tightly-Integrated Software Development Environments: The IPSEN Approach*. LNCS 1170. Springer-Verlag, Berlin, 1996.

[8] A. Schürr, A. Winter, and A. Zündorf. Graph grammar engineering with PROGRES. In W. Schäfer and P. Botella, editors, *Proceedings of the European Software Engineering Conference (ESEC '95)*, LNCS 989, pages 219–234, Barcelona, Spain, Sept. 1995. Springer Verlag.

[9] A. Sheth et al. NSF workshop on workflow and process automation. *ACM SIGSOFT Software Engineering Notes*, 22(1):28–38, Jan. 1997.

[10] B. Westfechtel. A graph-based system for managing configurations of engineering design documents. *International Journal of Software Engineering and Knowledge Engineering*, 6(4):549–583, Dec. 1996.

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