Graph-Based Models for Managing Development Processes, Resources, and Products

Carl-Arndt Krapp[1], Sven Krüppel[2], Ansgar Schleicher[3], and Bernhard Westfechtel[3]

[1] Finansys, Inc.
One World Trade Center, New York, NY 10048-0202
[2] SAP AG, Automotive Core Competence Center
Neurottstr. 16, D-69190 Walldorf, Germany
[3] Department of Computer Science III, Aachen University of Technology
D-52056 Aachen, Germany

Abstract. Management of development processes in different engineering disciplines is a challenging task. We present an integrated approach which covers not only the activities to be carried out, but also the resources required and the documents produced. Integrated management of processes, resources, and products is based on a model which is formally specified by a programmed graph rewriting system. Management tools are generated from the formal specification. In this way, we obtain a management system which assists in the coordination of developers cooperating in the development of a complex technical product.

1 Introduction

Development of products in disciplines such as mechanical, electrical, or software engineering is a challenging task. Costs have to be reduced, the time-to-market has to be shortened, and quality has to be improved. Skilled developers and sophisticated tools for performing technical work are necessary, yet not sufficient prerequisites for achieving these ambitious goals. In addition, the work of developers must be coordinated so that they cooperate smoothly. To this end, the steps of the development process have to be planned, a developer executing a task must be provided with documents and tools, the results of development activities have to be fed back to management which in turn has to adjust the plan accordingly, the documents produced in different working areas have to kept consistent with each other, etc.

Management can be defined as "all the activities and tasks undertaken by one or more persons for the purpose of planning and controlling the activities of others in order to achieve an objective or complete an activity that could not be achieved by the others acting alone" [27]. Management is concerned with processes, resources, and products:

- *Process management* covers the creation of process definitions, their instantiation to control development processes, as well as planning, enactment, and monitoring.

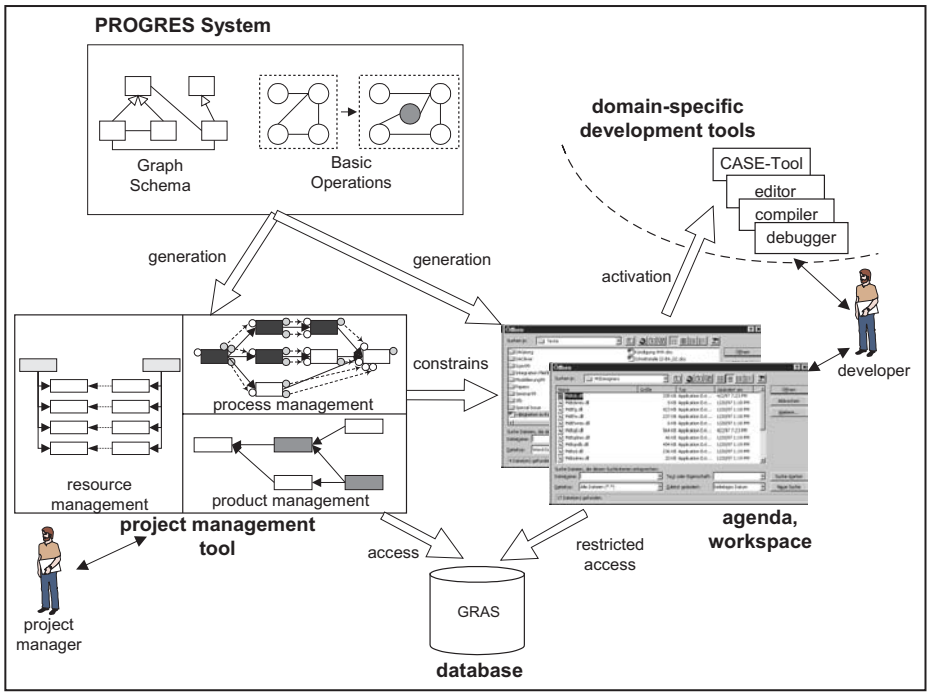

Fig. 1. A management system based on graph rewriting

- *Resource management* refers to both human and computer resources. In particular, it comprises the organization of human resources, the representation of computer resources, and the allocation of both human and computer resources to development activities.
- *Product management* deals with the documents created throughout the development life cycle, their dependencies, configurations of documents and dependencies, and versioning of both documents and configurations.

It is highly desirable to support both managers and developers through sophisticated tools assisting in the integrated management of processes, resources, and products. However, designing and implementing such tools constitutes a major challenge. In particular, the tools have to operate on complex data, and they have to offer complex commands manipulating these data.

We present a *programmed graph rewriting system* which serves as a high-level tool specification. Moreover, tools are generated from this executable specification. Our approach integrates the management of processes, resources, and products (for more detailed information on the corresponding submodels, see [12], [17], and [29], respectively). So far, we have applied our approach to three different domains, namely mechanical, chemical, and software engineering. In this paper, we will content ourselves to software engineering.

Figure 1 provides an overview of our *graph-based management system*. The management model is defined in PROGRES, which denotes both a language and an environment for the development of programmed graph rewriting systems [26]. Various kinds of tools are generated from the specification. A project management tool offers graphical views on management data and provides commands for planning and controlling development projects. Developers are supported by an agenda tool which displays a list tasks to be executed. Finally, a workspace tool is responsible for providing all documents and development tools required for executing a certain task.

The rest of this paper is structured as follows: Section 2 describes models for managing processes, resources, and products at an informal level. Section 3 presents cutouts of the corresponding PROGRES specifications. Section 4 discusses related work. Section 5 concludes the paper.

2 Management Models

2.1 Process Model

Our process management model is based on DYNAMIC Task nEts (*DYNAMITE* [11]), which support dynamic development processes through evolving *task nets*. Editing, analysis, and execution of task nets may be interleaved seamlessly. A task is an entity which describes work to be done. The interface of a task specifies what to do (in terms of inputs, outputs, pre- and postconditions, etc.). The realization of a task describes how to perform the work. A suitable realization may be selected from multiple alternative realization types. A realization is either atomic or complex. In the latter case, there is a refining subnet (task hierarchies). In addition to decomposition relationships, tasks are connected by control flows (similar to precedence relationships in net plans), data flows, and feedback flows (representing feedback in the development process).

To illustrate these concepts, Figure 2 shows the evolution of a task net for the development of a software system. At the beginning of a software development project only little is known about the process. A Design task is introduced into the net, while the rest of the task net remains unspecified as it is dependent on the design document's internal structure (part i). As soon as the coarse design (part ii) is available, the complete structure of the task net can be specified (part iii). For each module defined in the design document, three tasks for designing the interface, for implementing the body, and for testing are inserted into the task net. Control flow dependencies are established between the tasks, partly based on the product dependencies. Implementation tasks require export and import interfaces; moreover, modules are tested in a bottom-up order.

A control flow successor may be started before its predecessor terminates (*simultaneous engineering*). For example, design of module interfaces may commence before the coarse design is fixed in its final form (part iv). During detailed design, errors may be detected, raising *feedback* to the coarse design (from DesignIntB to Design). Subsequently, a new version of the design docu-

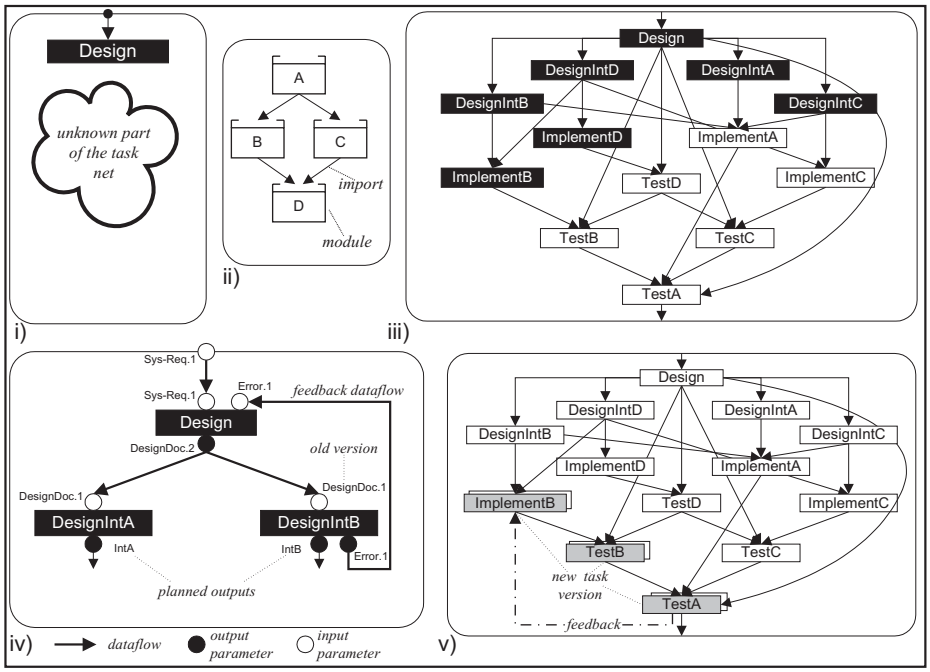

Fig. 2. Snapshot of a task net's evolution

ment (DesignDoc.2) is produced which will eventually be read by all successor tasks (so far, the successors are still using the old version).

If feedback occurs to terminated tasks, these are not reactivated. Rather, a new *task version* is derived from the old one and the old work context is reestablished (part v). This ensures traceability of the consequences of feedback. In our example, the first execution of TestA detects an error in module B. As a consequence, a new version of ImplementB is created. Version creation propagates through the task net until a new version of TestA is eventually created and executed.

2.2 Resource Model

RESMOD (*RES*ource Management *MOD*el, [17]) is concerned with the *resources* required for executing development processes. This includes human and computer resources, which are modeled in a uniform way. The notion of human resource covers all persons contributing to the development process, regardless of the functions they perform (e.g., managers as well as engineers). A similarly broad definition applies to computer resources, which cover all kinds of computer support for executing development processes, including hardware and software.

A resource is declared before it may be used in different contexts. A *resource declaration* introduces the name of a resource, its properties (attributes), and

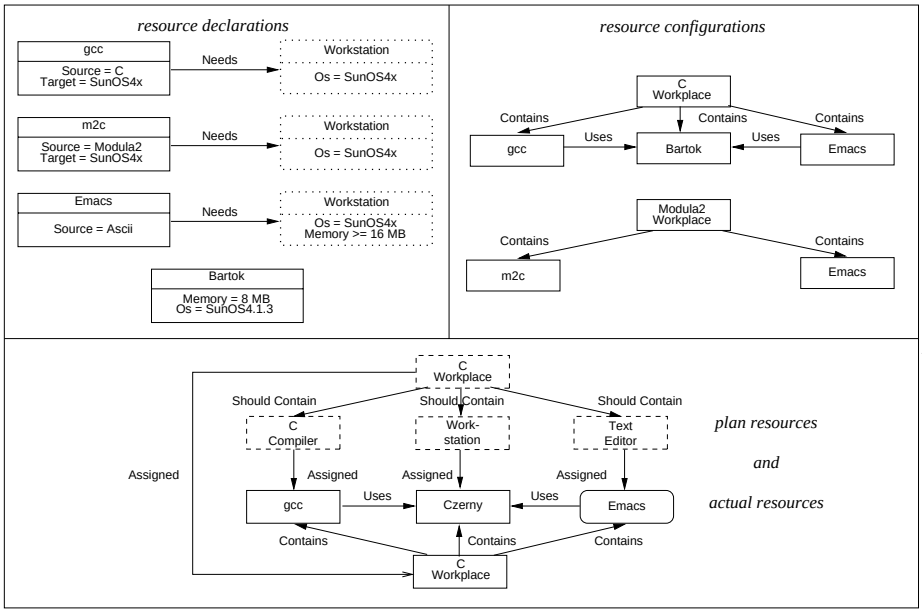

Fig. 3. Modeling of resources

(potentially) *needed resources*. For example, in Figure 3 the compiler gcc is characterized by its source language C and the target architecture SunOS4x. The compiler also needs a SunOS4x workstation, i.e., it runs on the same sort of machine for which it generates code. Note that the text editor Emacs needs at least 16 MB of main memory to execute with reasonable performance.

A *resource configuration* consists of a collection of resources. Resource configurations may be nested, i.e., a subconfiguration may occur as a component of some resource configuration. The resource hierarchy forms a directed acyclic graph. In general, a resource may be contained in multiple resource configurations. *Use relationships* connect inter-dependent components of resource configurations. These connections are established by binding formal needed resources to actual resources that have to meet the requirements stated in the resource declarations. Note that the resource configurations shown in Figure 3 are both incorrect. The first one is inconsistent because Bartok offers less main memory (8 MB) than required by Emacs. The second one is incomplete because it does not contain a workstation on which the tools may execute.

So far, we have considered *actual resources* and their relationships. In addition, RESMOD supports planning of resource requirements. A manager may specify in advance which resources he will need for a certain purpose, e.g., for running a development project. A *plan resource* serves as a placeholder for some actual resource. In the planning phase, the manager builds up configurations of plan resources. For each plan resource, he describes the requirements of a matching actual resource. Furthermore, a configuration of plan resources may

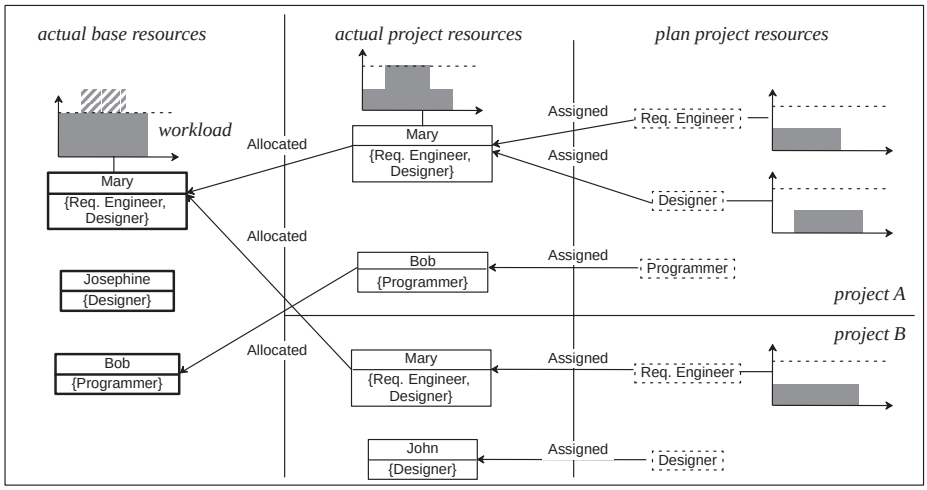

Fig. 4. Multi-project management

represent dependencies between its components. Later on, the manager assigns matching actual resources to the plan resources. In the simple example of Figure 3, C_Workplace represents a configuration of plan resources which are mapped 1:1 onto matching actual resources (in general, n:1 mappings are allowed).

Finally, RESMOD supports multi-project management (Figure 4). To this end, a distinction is made between *base resources* and *project resources*. A base resource belongs to the persistent organization of an enterprise and may be *allocated* to multiple projects. Alternatively, a resource may be acquired for a specific project only. The example given in Figure 4 demonstrates this for human resources. On the left-hand side, the available developers and their potential roles are described. For example, Mary may act as a requirements engineer or as a designer. On the right-hand side, each project manager specifies resource requirements in terms of plan resources and expected workloads. For each plan resource, a matching actual resource is either acquired specifically for this project (John), or it is drawn from the base organization (Mary). Workloads are accumulated both within a project (Mary fills multiple positions in project A) and across multiple projects (Mary is engaged in both A and B). In the example, the expected workload of Mary reveals an overload that requires rescheduling of resources.

2.3 Product Model

CoMa (*Co*nfiguration *Ma*nagement, [29]) supports version control, configuration control, and consistency control for heterogeneous documents through an integrated model based on a small number of concepts. In the course of development, documents such as requirements definitions, designs, or module bodies are created with the help of heterogeneous tools. Documents are related by man-

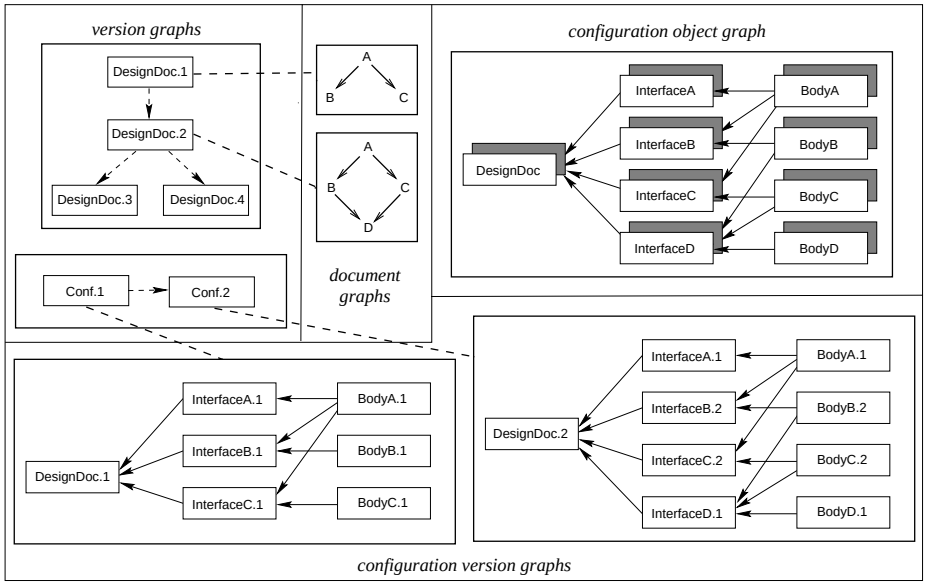

Fig. 5. Version and configuration graphs

ifold dependencies, both within one working area and across different working areas. The representation of these dependencies lays the foundations for consistency control between interdependent documents. Documents and their mutual dependencies are aggregated into configurations. Since development processes may span long periods of time, both documents and configurations evolve into multiple versions. Versions are recorded for various reasons, including reuse, backup, and provision of stable workspaces for engineers. Consistency control takes versioning into account, i.e., it is precisely recorded which versions of different documents are consistent with each other.

In order to represent objects, versions, and their relationships, the product management model distinguishes between different kinds of interrelated subgraphs (Figure 5). A *version graph* consists of versions which are connected by successor relationships. Versions are maintained for both documents and configurations. In our example, the evolution of the design of a software system is represented by a version tree. Each node refers to a *document graph* representing the contents of the respective version. Moreover, there is a version sequence representing the evolution of the overall product configuration. The contents of each version is contained in a corresponding configuration version graph.

A *configuration version graph* represents a snapshot of a set of interdependent components. Thus, it consists of component versions and their dependencies. In our example, the initial configuration version consists of initial versions of the design, module interfaces, and module bodies. In the second configuration version, the design was modified by adding a new module D. Interface and body of D were added as well; the interfaces and bodies of B and C were modified.

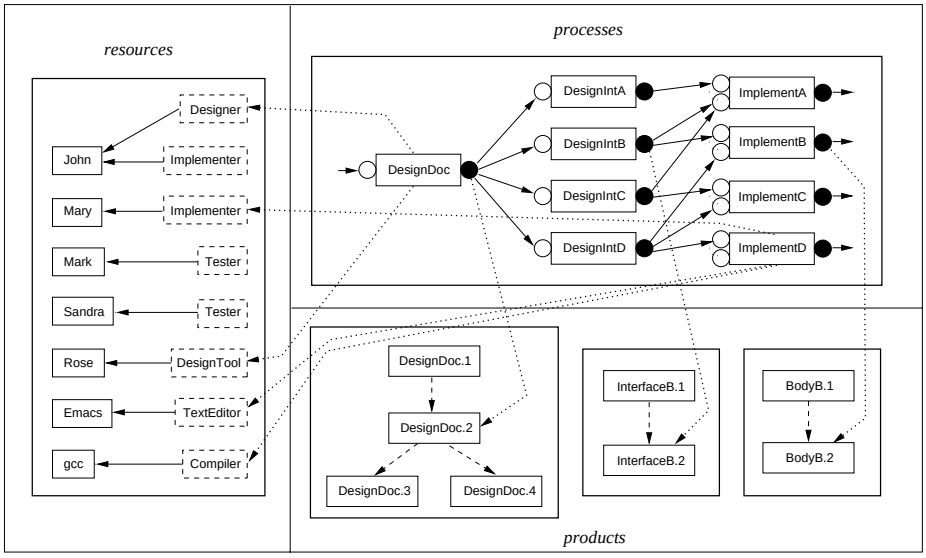

Fig. 6. Integration of processes, resources, and products

Finally, a *configuration object graph* represents version-independent structural information. It consists of documents which are connected by dependencies. For each component version (version dependency) in some configuration version graph, there must be a corresponding component object (object dependency) in the configuration object graph. Thus, the configuration object graph may be regarded as a "union" of all configuration version graphs.

2.4 Model Integration

The models for managing products, activities, and resources constitute components of an *integrated management model* (Figure 6). For example, the inputs and outputs of tasks refer to versions of documents or configurations. Furthermore, tasks are connected to both human and computer resources (responsible employees and supporting tools, respectively). Note that tasks are related to plan resources so that assignments can be made before the corresponding actual resources have been selected.

3 Formal Specification

Management models as presented in Section 2 consist of many entities which are mutually interrelated. Based on our experience in building structure-oriented environments, the data model of attributed graphs has proved suitable for the internal representation of complex data structures [19]. An *attributed graph* consists of attributed nodes which are interconnected by labeled, directed edges.

During editing, analysis, and execution of a management model, complex transformations and queries are performed on the internal data structure. We have chosen *programmed graph rewriting* in order to specify these complex operations on a high level of abstraction. In particular, we have used the specification language *PROGRES*, which is based on programmed graph rewriting [26].

Below, we will present small cutouts of the specification of the management submodels and their integration. Due to the lack of space, we will discuss only the *base models*, i.e., the model cores shared by all application domains. In order to use the models in different domains such as software, chemical, or mechanical engineering, they need to be enriched with domain-specific knowledge. For model customization, the reader is referred to [15, 24].

3.1 Specification of the Process Model

After having introduced the process management model informally in Subsection 2.1, we now turn to its formal specification. Let us first present the internal data structure maintained by a process management tool as presented in Figure 7 (see Figure 2, part iv for the corresponding external representation). It consists of typed nodes and directed edges which form binary relationships between nodes. A task graph as the internal data structure of a process model instance consists of nodes representing tasks, parameters and token as references to products maintained by a corresponding product model instance. Task relations and data flows are internally represented by nodes, because they carry attributes and neither PROGRES nor the underlying graph database support attributed edges. Edges are used to connect task, parameter and token nodes.

In order to restrict the graph to meaningful structures with respect to the process model, a *graph schema* is defined which specifies a graph type. The process model's graph schema is displayed in Figure 8. Node class `ITEM` serves as the root class. New versions can be derived from all elements of a task net (this ensures traceability; see Figure 2, part v). A successor version is reached by following the `toSuccessor` edge. On the next layer of the inheritance hierarchy we mainly distinguish between process entity and process relationship types. The node class `TOKEN` describes nodes representing tokens that are passed along data flows. `TASK` nodes own `PARAMETER` nodes which are either `INPUT` or `OUTPUT` parameters. Tasks can `produce` tokens via output parameters and `read` tokens via input parameters. Tasks are connected by `TASK RELATIONS` which can be vertical or horizontal relationships. `DECOMPOSITION` is a vertical task relation, instances of which are used to build a task hierarchy. Horizontal relationships are `CONTROL FLOW` or `FEEDBACK` relations. Parameters in turn are connected via `DATA FLOW` relationships which refine task relationships.

In addition to the graph schema, various *consistency constraints* are needed to define valid task net structures. For example, it has to be enforced that the task hierarchy must form a tree and that control flow relations do not contain cycles. PROGRES offers language constructs for graphical and textual constraint definition. Due to space restrictions we will not present formal constraint definitions here (but we will show below how graph rewrite rules check constraints).

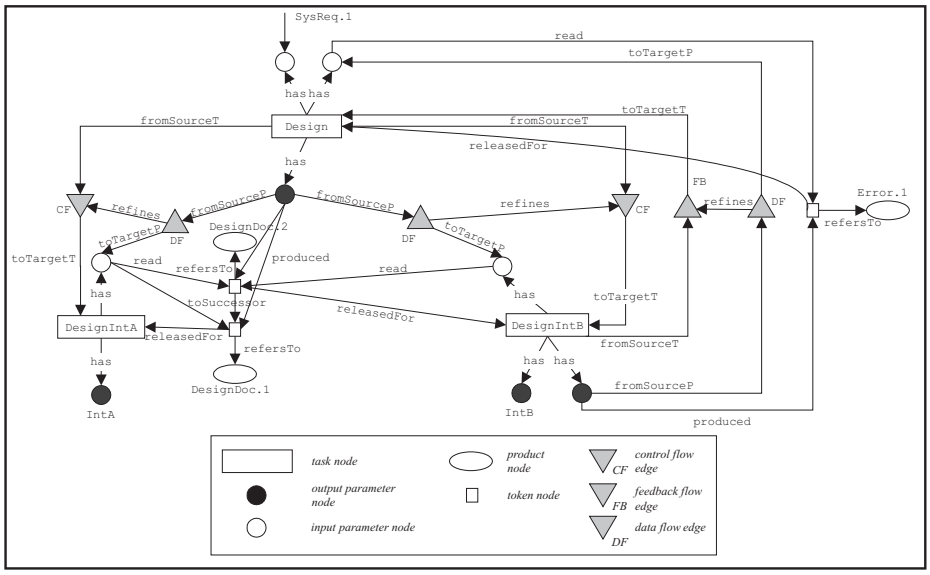

Fig. 7. Task graph

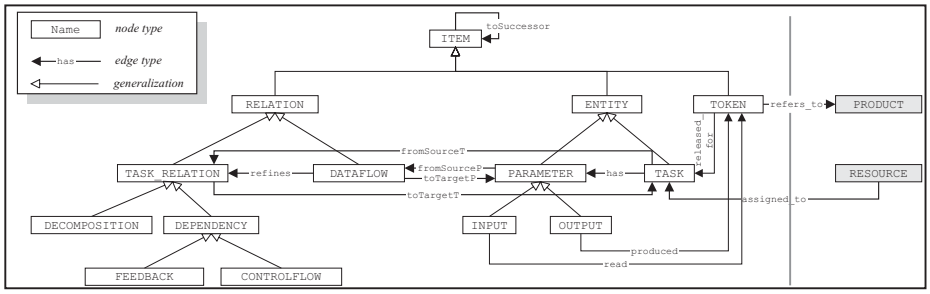

Fig. 8. The process model's graph schema

Rather, we will explain examples of the process model's operations which are divided into operations for *editing* and for *executing* a task net. Task nets can be edited by introducing new entities and relationships into the net. Equally well entities and relationships can be removed. Operations to execute a task net deal e.g. with the change of task states and the token game between tasks. Editing and execution of task nets are both described using a uniform mechanism. This allows to specify the intertwined editing and execution of task nets.

An example for an *execution operation* will be given in Subsection 3.4. As an example of an *edit operation* we chose a *graph rewrite rule* (*production* in PROGRES terminology) for feedback creation (Figure 9). The production is fed with the source and target tasks and the type of the feedback flow. In the left-hand side, a graph pattern is described that has to be found in order to create the feedback flow. Firstly, it has to be ensured that no feedback of the same type

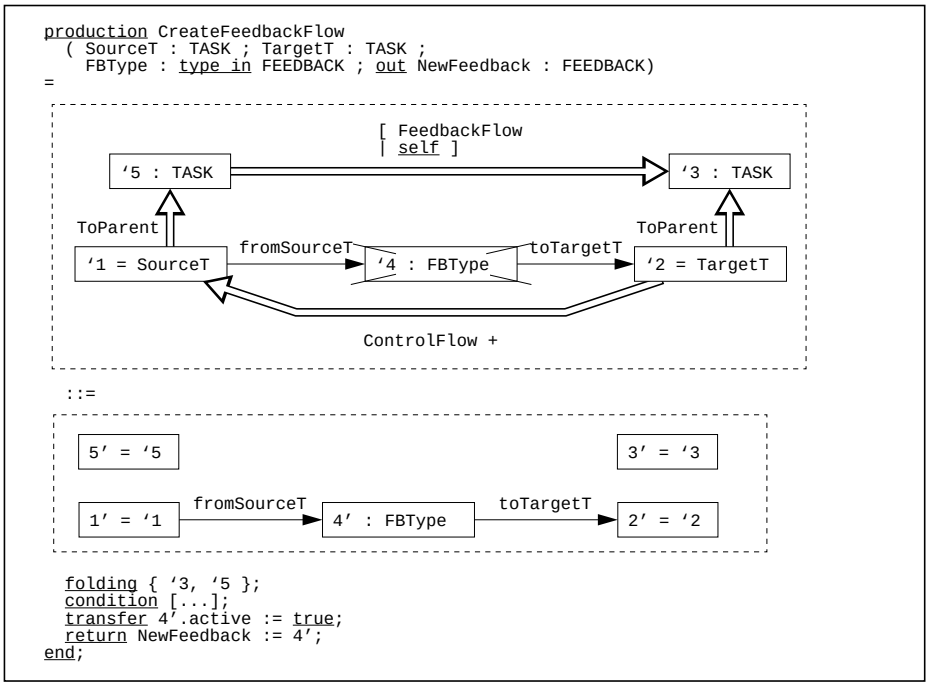

Fig. 9. Production for feedback flow creation

already exists between the two tasks. This is achieved by the negative node ‘4. Secondly, it has to be ensured that the feedback flow to be created is directed oppositely to control flows. When a feedback flow is created from a source to a target task (‘1 and ‘2), there must be a (transitive) control flow path in the opposite direction. Thirdly some restrictions have to hold on the parent tasks of the feedback's source and target. If the parent tasks are unequal (‘3 $\neq$ ‘5), they must be connected by a feedback flow as well. Otherwise, source and target share the same parent task (‘3 = ‘5). This is allowed by the folding clause above the (elided) condition part. In this case, the `FeedbackFlow` path from ‘5 to ‘3 collapses (alternative **`self`** in the path definition).

If the left-hand side could be matched in the current graph, a new feedback flow is created by the right-hand side (the other nodes occurring on the right-hand side are replaced identically). The created feedback flow is set to active in the transfer part of the production (the flow becomes inactive as soon as feedback has been processed). Finally, the node representing the created feedback flow is returned to the caller.

3.2 Specification of the Resource Model

A *resource graph* serves as the internal data structure maintained by the resource management tool. The types of nodes, edges, and attributes are defined in a

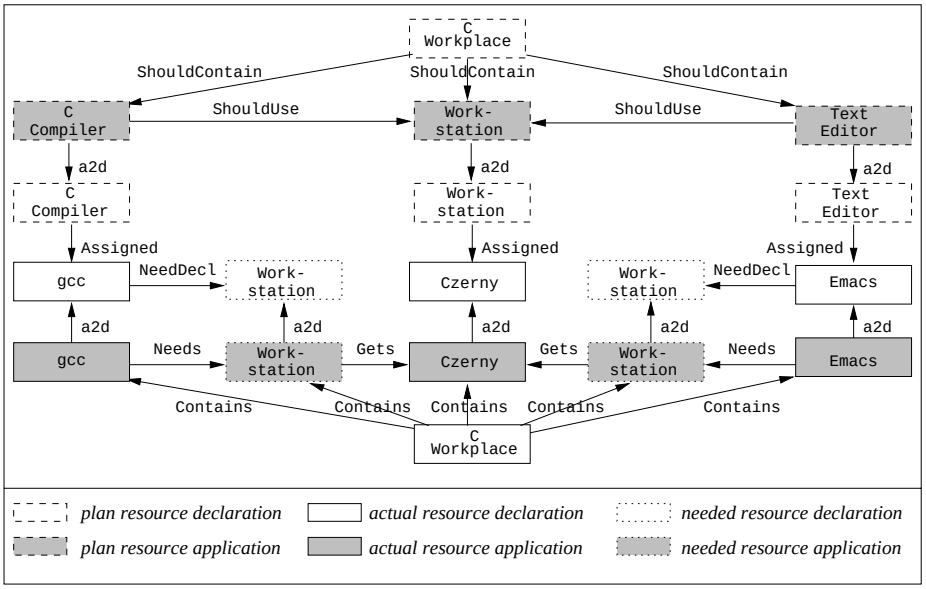

Fig. 10. Resource configurations

graph schema. Due to the lack of space, we refrain from introducing the graph schema; instead, we briefly discuss a small example.

Figure 10 shows the internal representation of the configurations of plan and actual resources depicted earlier in Figure 3. Nodes of different classes are distinguished by means of line and fill styles (see legend at the bottom); the strings written inside the boxes are formally represented as attributes (other attributes are not shown).

The internal graph model distinguishes between *resource declarations* and *resource applications* (white and grey boxes, respectively). In this way, it is possible to model context dependent applications. As a consequence, resource hierarchies are modeled by alternating declarations and applications. For example, the text editor `Emacs` needs some workstation as host. In the `C_Workplace` shown in the figure, the workstation `Czerny` has been selected. In a different context, the editor may be run on another host. Whether the resource requirements are met, is a context dependent property which is attached to the applied occurrence of the needed resource.

Figure 11 shows operations for building up *resource hierarchies*. The base operation `CreateSubresource` creates a node that represents an applied occurrence of some resource declaration (parameter `Subres`) in some resource configuration (`Res`). The negative path on the left-hand side ensures that no cycle is introduced into the resource hierarchy. Furthermore, we have to check whether the child resource is already contained in the parent. This is excluded by the negative node `‘3`: There must be no applied resource with the same name as `Subres` (see the restriction below the node).

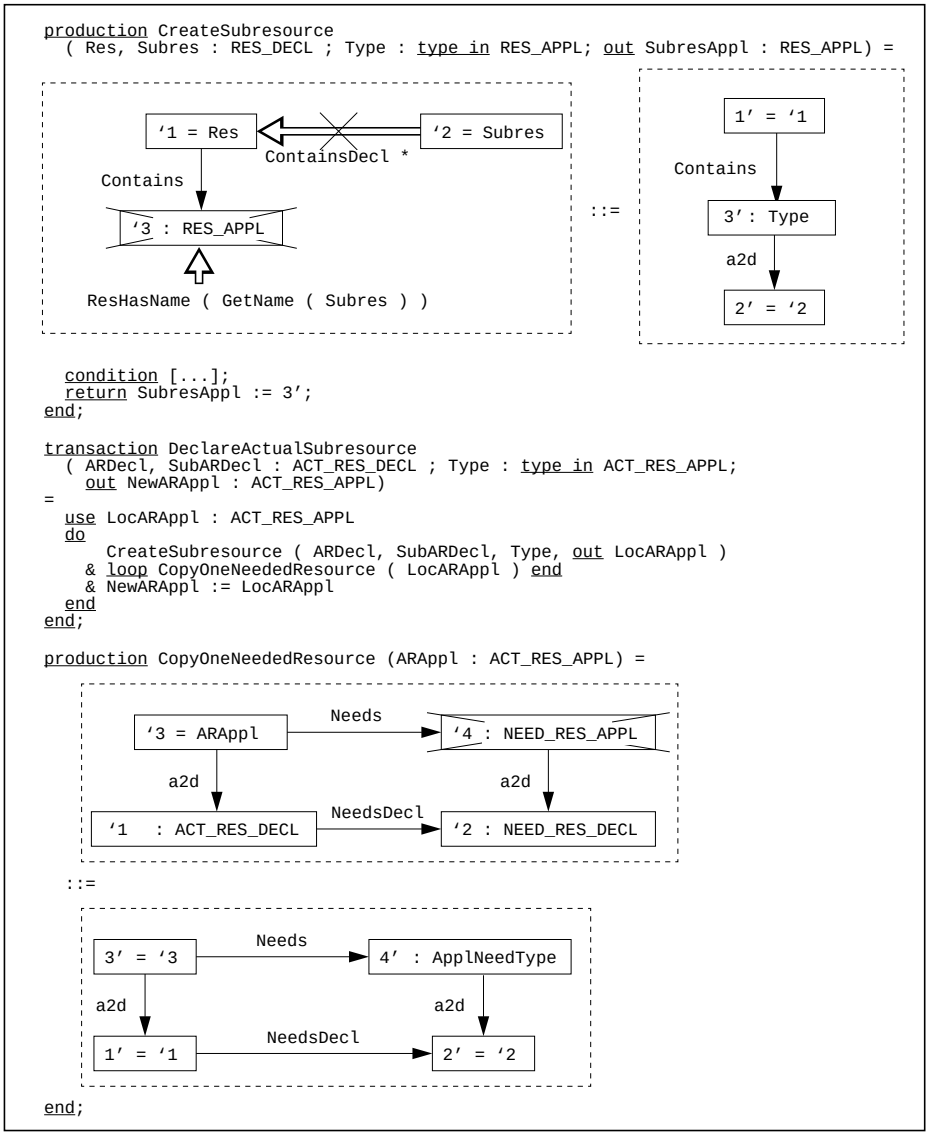

Fig. 11. Insertion of a subresource

The base operation does not take care of the needed resources, applied occurrences of which have to be created as well. Since this complex graph transformation cannot be expressed by a single graph rewrite rule, we compose multiple rules into a *transaction*. The transaction `DeclareActualSubresource` calls the base operation `CreateSubresource` and then performs a loop over all needed resources (the operator `&` and the keyword `loop` denote a sequence and a loop, respectively). `CopyOneNeededResToActualRes` creates an applied occurrence for

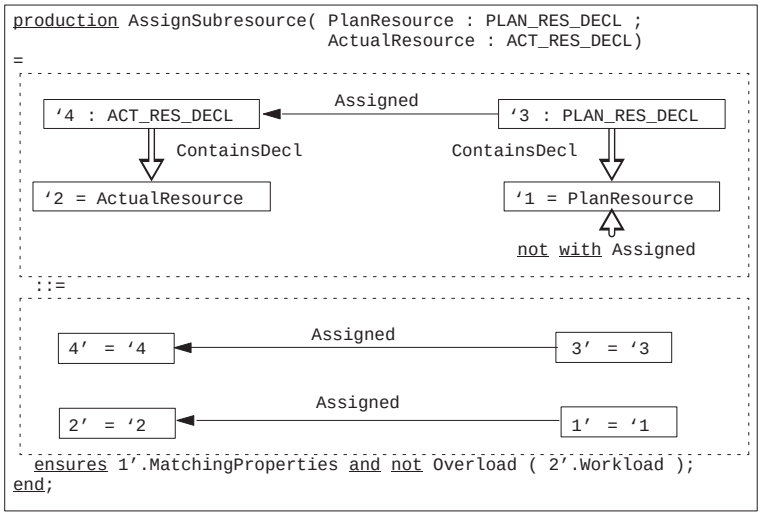

Fig. 12. Assignment of a subresource

one needed resource. The negative node on the left-hand side ensures that the applied occurrence has not been created yet, guaranteeing loop termination.

The rule shown in Figure 12 creates an *assignment relationship* between a plan resource and an actual resource (nodes `‘1` and `‘2` on the left-hand side, respectively). The rule may be applied only to subresources; assignments of root resources are handled by another rule. The restriction on node `‘1` ensures that no actual resource has been assigned yet to this plan resource (no outgoing `Assigned` edge). Moreover, the parents of nodes `‘1` and `‘2` must already have been connected by an assignment relationship; otherwise, resource assignments do not conform to resource hierarchies.

The **ensures** keyword below the right-hand side denotes a *postcondition* on attribute values. The rule fails if this postcondition does not hold. First, the actual resource must match the requirements attached to the plan resource. This is guaranteed when the attribute `MatchingProperties` evaluates to **true** (see below). Second, the resource assignment must not have caused an overload. This is checked by the predicate `Overload` that is applied to the attribute `Workload` of the actual resource.

How do we define the attributes used in the postcondition? The essential idea is to use *derived attributes* to this end (see [17] for further details). For example, if we have specified the resource requirements to some workstation, we want to make sure that the actual workstation meets these requirements. This can be handled as follows: Attributes `ReqMemorySize` and `ActMemorySize` are attached to the plan resource and the actual resource, respectively. The attribute `MatchingProperties` is then calculated by comparing these attributes:

```
MatchingProperties =
  self.ReqMainMemorySize <= self.Assigned.ActMainMemorySize
```

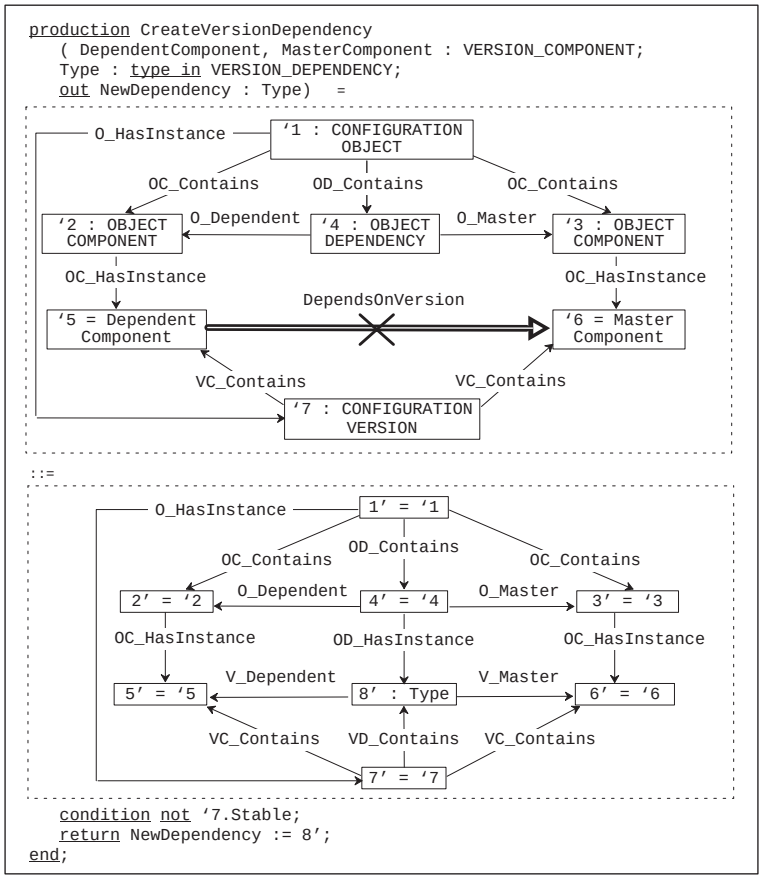

Fig. 13. Creation of a version dependency

3.3 Specification of the Product Model

The specification of the product model is sketched only briefly; for further information, the reader is referred to [29].

The internal representation of version graphs, configuration version graphs, and configuration object graphs is determined according to similar design rules as in the case of process or resource management. For each subgraph of the overall *product graph*, a root node is introduced. As in RESMOD, the problem of context dependence recurs: A version may occur in different configuration versions in different contexts. Therefore, we distinguish between applied occurrences — denoted as version components — and declarations (of versions). Finally, relationships are internally represented by nodes if they are decorated with attributes or relationships between relationships have to be represented.

An example of a transformation of the product graph is given in Figure 13. The graph rewrite rule creates a version dependency (node `8'`) between a depen-

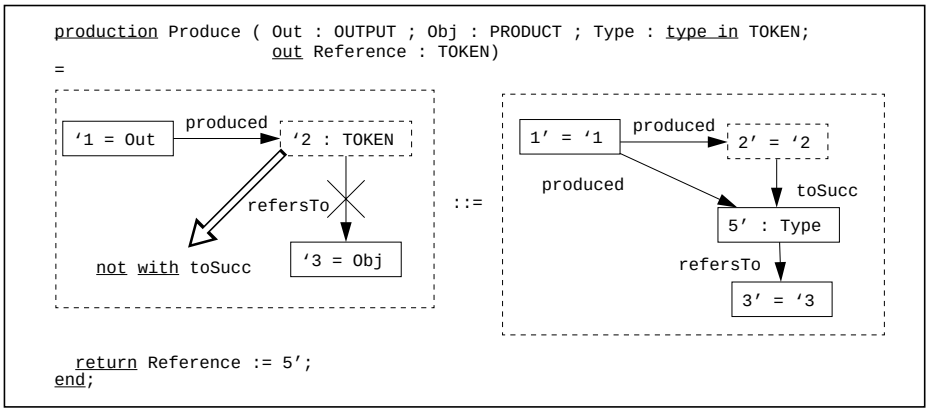

Fig. 14. Production of an output

dent component (‘5) and a master component (‘6) in a configuration version graph (‘7). Several constraints have to be checked by this rule: The configuration version must not have been frozen yet (condition part). Furthermore, master and dependent component must belong to the same subgraph (edges from ‘5 and ‘6 to ‘7). Finally, there must be a corresponding object dependency (‘4) in the configuration object graph (‘1). Please recall that a configuration object graph serves as an abstraction over a set of configuration version graphs (see Subsection 2.3).

3.4 Specification of Model Integration

Due to the lack of space, we discuss model integration only briefly. We confine ourselves to the integration between process and product model. Concerning process and resource integration, we refer the reader back to Subsection 3.2 because the assignment of plan resources to tasks can be handled similarly to the assignment of actual to plan resources (Figure 12).

With respect to model integration, we favor a *loose coupling* between the submodels. That is, we connect two submodels such that the modification of one submodel has minimal impact on the other one. This is illustrated by the graph rewrite rule `Produce`, which belongs to the execution operations of DYNAMITE (Figure 14). To establish a clear separation between the process model and the product model, the output of some task does not refer to some object directly. Rather, a *token* is created which is connected to the output parameter on one hand and the produced object on the other hand. In the DYNAMITE specification, it is merely assumed that there is a node class `PRODUCT` from which subclasses are derived in the product model. Note that all tokens are arranged in a sequence (`toSucc` edges), which is required for simultaneous engineering (see Subsection 2.1). Node ‘2 is optional, i.e., the graph rewrite rule either creates the first token or appends a token at the end of the list.

4 Related Work

DYNAMITE is based on instance-level task nets, where tasks and relations are dynamically instantiated from types. In general, the structure of a task net is known only at run time. In contrast, process-centered software engineering environments such as Process Weaver [9] or SPADE [1] are based on populated copies, i.e., a template of a Petri net is copied, populated with tokens, and enacted. This implies that the net structure is already determined at modeling time. Rule-based systems such as e.g. Marvel [14] or MERLIN [22] are more flexible in that respect. Such systems may dynamically generate plans from facts and rules, e.g. for compiling and linking program modules. While plans are maintained automatically and are normally hidden from the users, task nets are manipulated by the project manager.

RESMOD introduces fairly general concepts for resource management (resource configurations, plan and actual resources, base and project resources). These concepts can be applied to model a wide variety of organizational models. In contrast, many other systems implement a more specific model which lacks the required generality. This statement applies e.g. to the workflow management systems Leu [4], Flow Mark [13], and ORM [23]. Only MOBILE [3] is different in that it offers a very general base model for resource management. However, we believe that MOBILE is a bit too general and fails to fix anything specific to resource management. In MOBILE, a resource management model must be essentially specified from scratch, starting from general entity/relationship types.

CoMa integrates version control, configuration control, and consistency control. To this end, graphs are used to represent the complex structures encountered in product management. In particular, this distinguishes CoMa from the main stream software configuration management systems, which essentially add version control to the file system [28, 18]. Concerning the relationships between versioned objects, most existing systems focus on hierarchies. This does not apply only to the file-based SCM systems mentioned above. In addition, systems relying on databases such as PCTE [21] or DAMOKLES [5] focus on versions of complex objects. In contrast, CoMa puts strong emphasis on dependencies among components of configurations. Thus, a configuration is modeled as a graph of components related by dependencies rather than just a directory.

PROGRES has been used in our group for a long time. PROGRES is a comprehensive specification language offering declarations of node types, node attributes, and edge types, derived attributes and relationships, constraints, graph tests, graph rewrite rules, control structures for combining these rules into transactions, and backtracking. In our specifications, we have exploited virtually all of these features. Moreover, we heavily rely on the PROGRES environment when building management tools. Given our requirements, the alternatives are not numerous. For example, the AGG environment [8] is based on the algebraic approach to graph rewriting, which is too restrictive for our applications. Recently, the Fujaba environment [10] has been developed which offers a graph grammar language UML [2]. Fujaba misses some essential features provided by PROGRES (in particular, derived attributes/relationships and backtracking).

5 Conclusion

We have presented an integrated model for managing development processes, resources, and products. Furthermore, we have formally specified this model by a programmed graph rewriting system. In this paper, we have presented cutouts of the formal specification of the management model. The total size of this specification is about 200 pages (counting only the base specifications).

While the examples given in this paper were taken from the software engineering domain, we have also studied applications in other engineering disciplines. A predecessor of the management model was developed in the SUKITS project, which dealt with development processes in mechanical engineering [30]. Current work is performed in the Collaborative Research Centre IMPROVE, which investigates development processes in chemical engineering [20].

References

1. S. Bandinelli, A. Fuggetta, and C. Ghezzi. Software process model evolution in the SPADE environment. *IEEE Transactions on Software Engineering*, 19(12):1128–1144, Dec. 1993.
2. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1998.
3. C. Bußler and S. Jablonski. An approach to integrate workflow modeling and organization modeling in an enterprise. In *Proceedings of the Third Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 81–95, Morgantown, West Virginia, Apr. 1994.
4. G. Dinkhoff, V. Gruhn, A. Saalmann, and M. Zielonka. Business process modeling in the workflow management environment Leu. In P. Loucopoulos, editor, *Proceedings of the 13th International Conference on Object-Oriented and Entity-Relationship Modeling (ER '94)*, LNCS 881, pages 46–63, Manchester, UK, Dec. 1994.
5. K. Dittrich, W. Gotthard, and P. Lockemann. DAMOKLES, a database system for software engineering environments. In R. Conradi, T. M. Didriksen, and D. H. Wanvik, editors, *Proceedings of the International Workshop on Advanced Programming Environments*, LNCS 244, pages 353–371, Trondheim, June 1986.
6. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, volume 2. World Scientific, Singapore, 1999.
7. G. Engels and G. Rozenberg, editors. *TAGT '98 — 6th International Workshop on Theory and Application of Graph Transformation*, technical report tr-ri-98-201, Paderborn, Germany, Nov. 1998.
8. C. Ermel, M. Rudolf, and G. Taentzer. The AGG approach: Language and environment. In Ehrig et al. [6], pages 551–602.
9. C. Fernström. PROCESS WEAVER: Adding process support to UNIX. In *Proceedings of the 2nd International Conference on the Software Process*, pages 12–26, Berlin, Germany, Feb. 1993.
10. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph grammar language based on the unified modeling language and Java. In Engels and Rozenberg [7], pages 112–121.

11. P. Heimann, C.-A. Krapp, and B. Westfechtel. An environment for managing software development processes. In *Proceedings of the 8th Conference on Software Engineering Environments*, pages 101–109, Cottbus, Germany, Apr. 1997.
12. P. Heimann, C.-A. Krapp, B. Westfechtel, and G. Joeris. Graph-based software process management. *International Journal of Software Engineering and Knowledge Engineering*, 7(4):431–455, Dec. 1997.
13. IBM, Böblingen, Germany. *IBM FlowMark: Modeling Workflow*, Mar. 1995.
14. G. E. Kaiser, P. H. Feiler, and S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, 5(3):40–49, May 1988.
15. C.-A. Krapp. *An Adaptable Environment for the Management of Development Processes.* Number 22 in Aachener Beiträge zur Informatik. Augustinus Buchhandlung, Aachen, Germany, 1998.
16. S. Krüppel. Ein Ressourcenmodell zur Unterstützung von Software-Entwicklungsprozessen. Master's thesis, RWTH Aachen, Germany, Feb. 1996.
17. S. Krüppel and B. Westfechtel. RESMOD: A resource management model for development processes. In Engels and Rozenberg [7], pages 390–397.
18. D. Leblang. The CM challenge: Configuration management that works. In W. Tichy, editor, *Configuration Management*, volume 2 of *Trends in Software*, pages 1–38. John Wiley & Sons, New York, 1994.
19. M. Nagl, editor. *Building Tightly-Integrated Software Development Environments: The IPSEN Approach.* LNCS 1170. Springer-Verlag, Berlin, Germany, 1996.
20. M. Nagl and W. Marquardt. SFB-476 IMPROVE: Informatische Unterstützung übergreifender Entwicklungsprozesse in der Verfahrenstechnik. In M. Jarke, K. Pasedach, and K. Pohl, editors, *Informatik '97: Informatik als Innovationsmotor*, Informatik aktuell, pages 143–154, Aachen, Germany, Sept. 1997.
21. F. Oquendo, K. Berrado, F. Gallo, R. Minot, and I. Thomas. Version management in the PACT integrated software engineering environment. In C. Ghezzi and J. A. McDermid, editors, *Proceedings of the 2nd European Software Engineering Conference*, LNCS 387, pages 222–242, Coventry, UK, Sept. 1989.
22. B. Peuschel, W. Schäfer, and S. Wolf. A knowledge-based software development environment supporting cooperative work. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):79–106, Mar. 1992.
23. W. Rupietta. Organization models for cooperative office applications. In D. Karagiannis, editor, *Proceedings of the 5th International Conference on Database and Expert Systems Applications*, LNCS 856, pages 114–124, Athens, Greece, 1994.
24. A. Schleicher, B. Westfechtel, and D. Jäger. Modeling dynamic software processes in UML. Technical Report AIB 98-11, RWTH Aachen, Germany, 1998.
25. A. Schürr and A. Winter. UML packages for programmed graph rewriting systems. In Engels and Rozenberg [7], pages 132–139.
26. A. Schürr, A. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In Ehrig et al. [6], pages 487–550.
27. R. H. Thayer. Software engineering project management: A top-down view. In R. H. Thayer, editor, *Tutorial: Software Engineering Project Management*, pages 15–54. IEEE Computer Society Press, Washington, D.C., 1988.
28. W. F. Tichy. RCS – A system for version control. *Software–Practice and Experience*, 15(7):637–654, July 1985.
29. 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.
30. B. Westfechtel. Graph-based product and process management in mechanical engineering. In Ehrig et al. [6], pages 321–368.