Chapter 1

GRAPH-BASED PRODUCT AND PROCESS MANAGEMENT IN MECHANICAL ENGINEERING

BERNHARD WESTFECHTEL
Lehrstuhl für Informatik III, RWTH Aachen
D-52056 Aachen, Germany

Development of products in disciplines such as mechanical, electrical, or software engineering is a challenging task. Thus, sophisticated tools for supporting the management of development products, processes, and resources are urgently needed. We report on a management system which we designed and implemented in a joint project with mechanical engineers. The system was developed from a high-level formal specification based on programmed graph rewriting. We demonstrate that graphs are indeed very well suited for representing management data such version histories, configurations, and task nets. In addition, graph rewrite rules provide high-level descriptions of complex graph transformations for the functions offered by the management system. The formal specification of the management model serves as a non-trivial case study demonstrating an important practical application of graph rewriting.

Contents

1.1 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 meeting 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" [1]. Management is concerned with products, processes, and resources:

- *Product management* deals with the documents created throughout the development life cycle, their dependencies, configurations composed of documents and dependencies, and versioning of both documents and configurations.
- *Process management* covers the creation of process definitions, their instantiation to control development processes, as well as planning, enactment, and monitoring.
- *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.

Numerous *tools* have been developed to support management. Usually, they are called x management tools (or systems, i.e., collections of tools), where x denotes the objects to be managed. Typical values of x are "engineering data" [2], "product data" [3], "software configuration" [4], "workflow" [5], "software process" [6], and "project" [7].
Initially, small tools were realized which provided support for specific problems, e.g., SCCS [8] for maintaining versions of text files and Make [9] for maintaining consistency between source and derived objects (e.g. for controlling the execution of compile and link steps in the case of large programs composed

of many files). Over the years, the tools became more and more complex, and they were combined into integrated systems of significant size (consider e.g. the software configuration management system ClearCase [10], the workflow management system FlowMark [11], or the software process management system PROCESS WEAVER [12]).

Unfortunately, many of these systems have been implemented in an ad hoc manner. Thus, it is by no means easy to figure out what kinds of management data (objects, relationships, and attributes) are maintained, which constraints they have to fulfill, how commands offered to the user affect these data, etc. In other words, the underlying *models* are poorly defined. A formal description of these at a high level of abstraction would not only be helpful in understanding the functionality of a given system. Rather, it also makes the construction of new systems significantly easier.

In particular, *graph rewriting* can serve these needs very well. Graphs are ideally suited for representing management data such as version histories, configurations of interdependent documents, and task nets. Operations on these graphs can be specified in a declarative way by means of graph rewrite rules. Since such a specification is executable, we may even generate tools from the specification.

In this chapter, we present a non-trivial *case study* for the practical application of graph rewriting. We report on work carried out in the *SUKITS* project [13]. SUKITS was a joint effort of mechanical engineers and computer scientists to improve computerized support for managing development processes. The project was funded by the Deutsche Forschungsgemeinschaft from 1991 to 1997. Within the SUKITS project, we designed and implemented a system for the integrated management of products, processes, and resources. The data and functions of this system are formally specified by a *programmed graph rewriting system*. We will mainly focus on this formal specification, parts of which have been published previously [14,15]. Our main intent is to demonstrate that graph rewriting is in fact well suited for the specification of management tools (as one important application domain). For further information on SUKITS, the reader is referred to [13,16,17,18].

In Section 1.2, we give a brief overview of the SUKITS project. Sections 1.3 and 1.4 present the SUKITS management model at an informal and at a formal level, respectively. Section 1.5 summarizes our contributions and gives an outlook on future work.

1.2 The SUKITS Project

SUKITS focuses on *mechanical engineering* as its application domain. More specifically, we studied the development of metal and plastic parts. Development subsumes in particular design, manufacturing planning, NC programming, and simulation. In these working areas, we strived for improving computerized support by designing and implementing a system for managing products, processes, and resources (Section 1.2.1). The management system is based on a programmed graph rewriting system acting as a formal specification (Section 1.2.2).

1.2.1 Management System

The management system, which is centered around a graph-based *management database*, supports both managers and engineers. To this end, it provides environments customized to the needs of different classes of users. Here, the term "environment" denotes a collection of software tools.
The *management environment* consists of tools which support project managers. In particular, a project manager may plan development tasks and super-

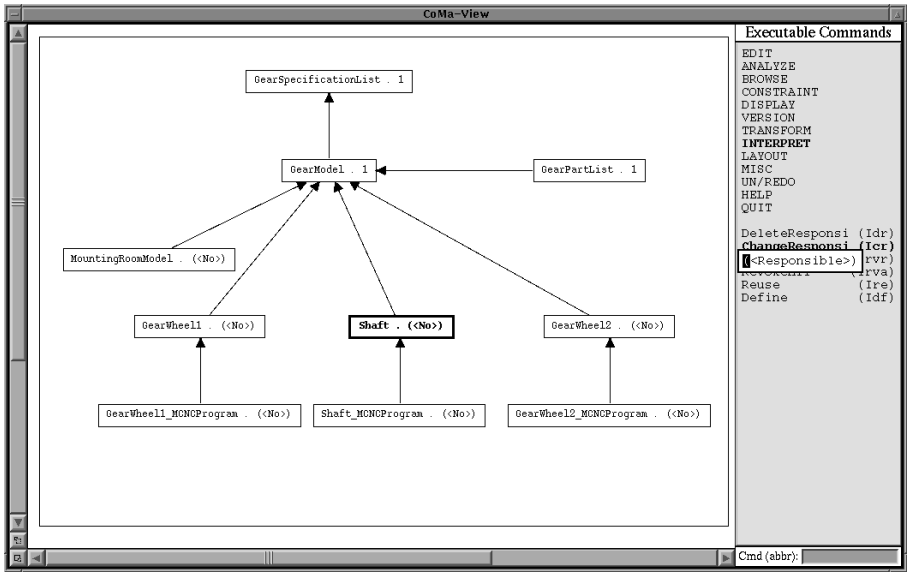


Figure 1.1: Management environment

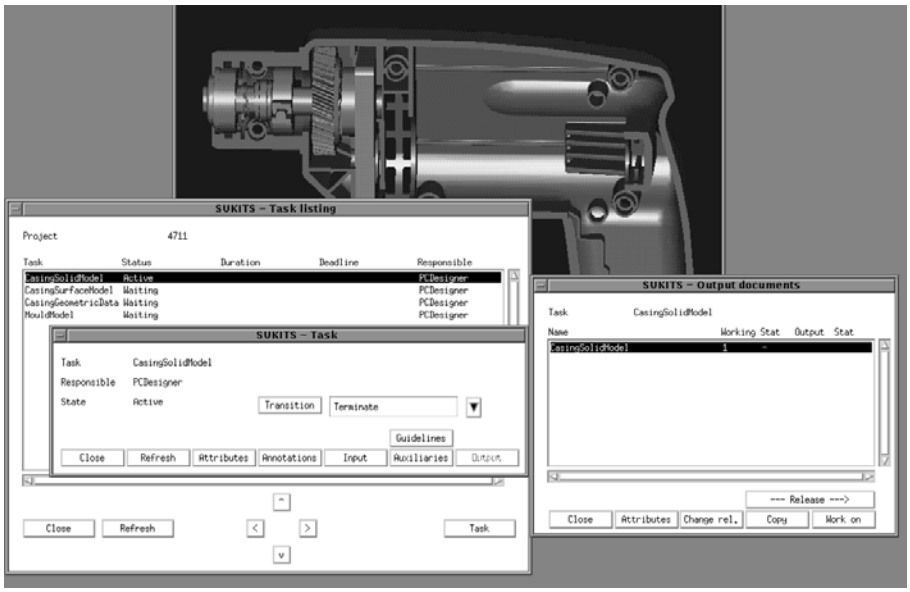


Figure 1.2: Work environment

vise their execution. Furthermore, he may define the project team and assign roles to its members. Finally, the management environment also includes tools for version and configuration management.

The user interface of the management environment is illustrated by the snapshot shown in Figure 1.1. The snapshot is taken from a session demonstrating the development of a drill. The window shows a configuration of interdependent documents for the gear of the drill. The manager has selected the component `Shaft` (bold face) and is just assigning the responsibility for designing the shaft to some engineer (input window containing the text `(<Responsible>)` on the right-hand side).

The *work environment* supports engineers by displaying agendas of assigned tasks and by establishing a detailed work context for each task (including input and output documents as well as the tools to operate on these documents). Tools such as CAD systems, CAP systems, or NC programming systems are invoked uniformly via the work environment. In this way, a posteriori integration of already existing tools is supported.

The work environment is illustrated by the snapshot of Figure 1.2. The large window on the left displays an agenda of tasks from which the engineer has

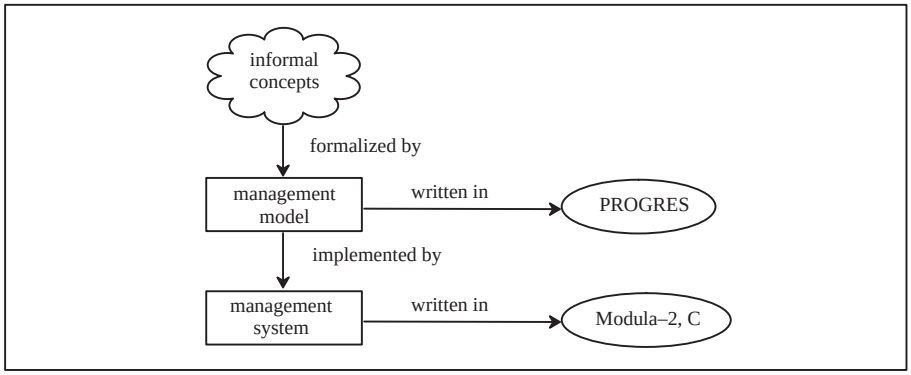


Figure 1.3: Basic notions and their interrelations

selected the design of the drill's casing. In the small window popping up in response to this selection, he has pushed the `Output` button to view the output documents (right-hand side). From there, he has started the CAD system for working on the solid model of the casing (background).

1.2.2 Management Model

The management system was specified formally in *PROGRES* [19,20,21,22], a language which is based on programmed graph rewriting (Figure 1.3). PROGRES is an attractive choice for the following reasons:

- PROGRES is a rich language combining multiple paradigms. It supports rule-based programming by means of graph rewrite rules and procedural programming by control structures. Moreover, all operations are checked against a graph schema (known from database programming languages) which in particular allows for the definition of derived data and constraints (for further details, see Chapter **??** of this volume).
- Currently, there are only a few operational specification environments based on graph rewriting (in addition to PROGRES, we may mention e.g. AGG, see Chapter **??** of this volume). Indeed, PROGRES denotes not only a language, but also a fairly sophisticated environment featuring tightly integrated tools for editing, analyzing, interpreting, and compiling specifications.

We are aware of only a few approaches applying graph rewriting to the management of development (e.g., [23,24]). This is the case even though graphs model

the data structures in this application domain in a uniform and natural way and graph rewrite rules allow for the high-level specification of graph transformations. Let us briefly compare some competing specification approaches (without striving for completeness):

- *Abstract syntax trees* have been used in various syntax-aided software development environments (or generators for such environments) such as e.g. Gandalf [25] and CPSG [26]. All of these environments mainly support programming. Abstract syntax trees seem a quite natural choice for the representation of programs, in particular because they have been exploited successfully in compiler construction. However, management of development processes has to deal with graphs rather than trees (e.g., version graphs, configuration graphs, and task graphs).

- Database systems such as DAMOKLES [27], PRIMA [28] and PCTE [29][a], which have been developed specifically for engineering applications, are based on *EER models* (which ultimately have their roots in the ER model proposed by Chen [30]). EER models may be used to represent graph-like structures; in this respect, they are superior to abstract syntax trees. Moreover, they provide built-in support for complex objects and often even for versions. Compared to PROGRES, the main weakness of most EER-based systems refers to the specification of updates. In PROGRES, graph rewrite rules are used to this end, while in EER-based systems updates have to be broken down to elementary operations for deleting/inserting/changing single entities/relationships/attributes. Note that this statement also applies to database systems which are based on an *object-oriented data model* (e.g., O_2 [31] and Objectstore [32]).

- Process management systems such as PROCESS WEAVER [12], MELMAC [33], and SPADE [34] are based on different variants of *Petri nets*. A process is modeled as a hierarchical collection of Petri nets. To prepare execution, a template is copied and populated with tokens. Execution is then modeled by the well-known token game. While Petri nets provide a formal foundation for analysis (e.g., deadlock and liveness) and execution (firing of transitions), editing (e.g., insertion/deletion of transitions/places) has to be described outside the Petri net formalism. In contrast, we define editing, analysis, and execution in a uniform formal framework. Moreover, our specification approach also covers products and resources, which are beyond the scope of Petri nets.

[a] Here, we refer to the object management system of PCTE.

1.3 Management Model: Informal Description

The management system is based on a model which is described informally in the current section and will be formalized in the next section. Because of space limitations, we discuss only the submodels for product and process management; for resource management, the reader is referred to [16].

1.3.1 Product Management Model

Documents, dependencies, and configurations 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. Physically, a document is typically represented as a file or a complex object in an engineering database.
Documents are related by manifold *dependencies*. Such dependencies may either connect documents 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.
Related documents (e.g., all documents describing a single part) are aggregated into *configurations*. In addition to components, configurations contain dependencies as well. Documents (*atomic objects* at the coarse-grained level) and configurations (*complex objects*) are treated uniformly. Vertical relationships between configurations and their components are called *composition relationships*. In particular, a component of some configuration may be a subconfiguration (nested configurations). Therefore, dependencies between subconfigurations can be modeled in the same way as dependencies between documents.

Versions and versioned objects A *version* v represents a state of an evolving object o. The latter is called *versioned object* (or simply object). A versioned object serves as a container of a set of related versions and provides operations for version retrieval and creation of new versions. We distinguish between *document objects* and *configuration objects*, whose versions are called *document versions* and *configuration versions*, respectively.

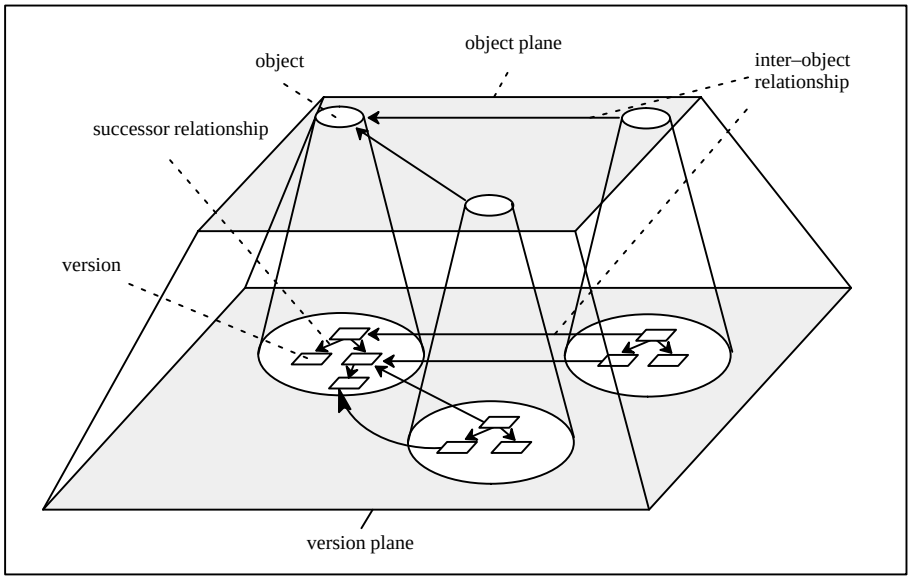


Figure 1.4: Object and version plane

According to the distinction between objects and their versions, the product management model distinguishes between an *object plane* and a *version plane* (Figure 1.4). The version plane refines the object plane: Each object is refined into its versions, and each relationship between two objects is refined into relationships between corresponding versions. Furthermore, successor relationships between versions of one object represent its evolution. While the object plane provides an overview by abstracting version-independent structural information, the version plane accurately represents actual versions of documents and configurations as well as their mutual relationships.

Version and configuration graphs Objects, versions, and their relationships are formally represented by *graphs*. In order to structure the information contained in such graphs, the product management model distinguishes different kinds of interrelated subgraphs, namely, version graphs, configuration version graphs, and configuration object graphs.
A *version graph* (Figure 1.5) consists of versions which are connected by *successor relationships*. A successor relationship from v_1 to v_2 indicates that v_2 was derived from v_1 (usually by modifying a copy of v_1). In simple cases, versions are arranged in a sequence reflecting the order in which they were created.

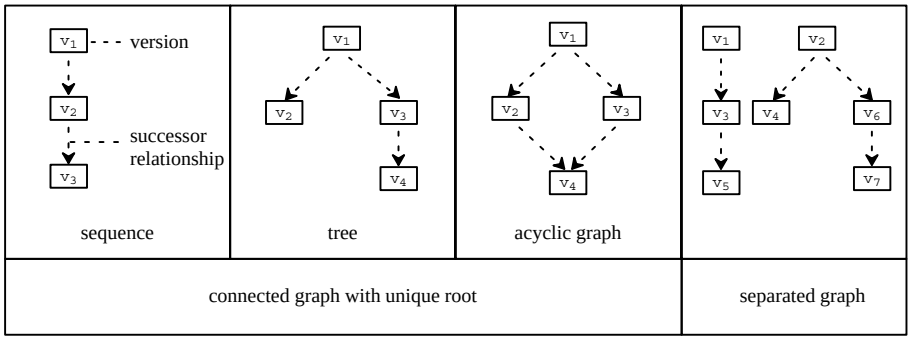


Figure 1.5: Structures of version graphs

Concurrent development of multiple versions causes branches in the evolution history (version tree). For example, v_2 may have been created to fix a bug in v_1, while work is already in progress to produce an enhanced successor v_3. 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. The product management model allows for all structures shown in Figure 1.5; it merely excludes cycles in successor relationships. Variants (alternative versions) may be represented either by branches or by separated subgraphs.

A *configuration version graph* represents a snapshot of a set of interdependent components. Thus, it consists of component versions and their dependencies. The lower part of Figure 1.6 shows three versions of a configuration `Shaft` which comprises all documents describing a shaft. Versions are denoted by the name of the versioned object, a dot, and a version number. The initial configuration version contains versions of a design, a manufacturing plan, and two NC programs (e.g., for turning the left and the right half of the shaft, respectively). The manufacturing plan depends on the design, and the NC programs depend on both the design and the manufacturing plan. In the second configuration version, the geometry of the shaft was modified, and accordingly all dependent documents were modified, as well. In the last version, it was decided to produce the shaft from a specifically designed raw part in order to minimize consumption of material (previously, a piece of some standard metal bar was used). To this end, a raw part design was added to the configuration, and the dependencies were extended accordingly.

A *configuration object graph* represents version-independent structural infor-

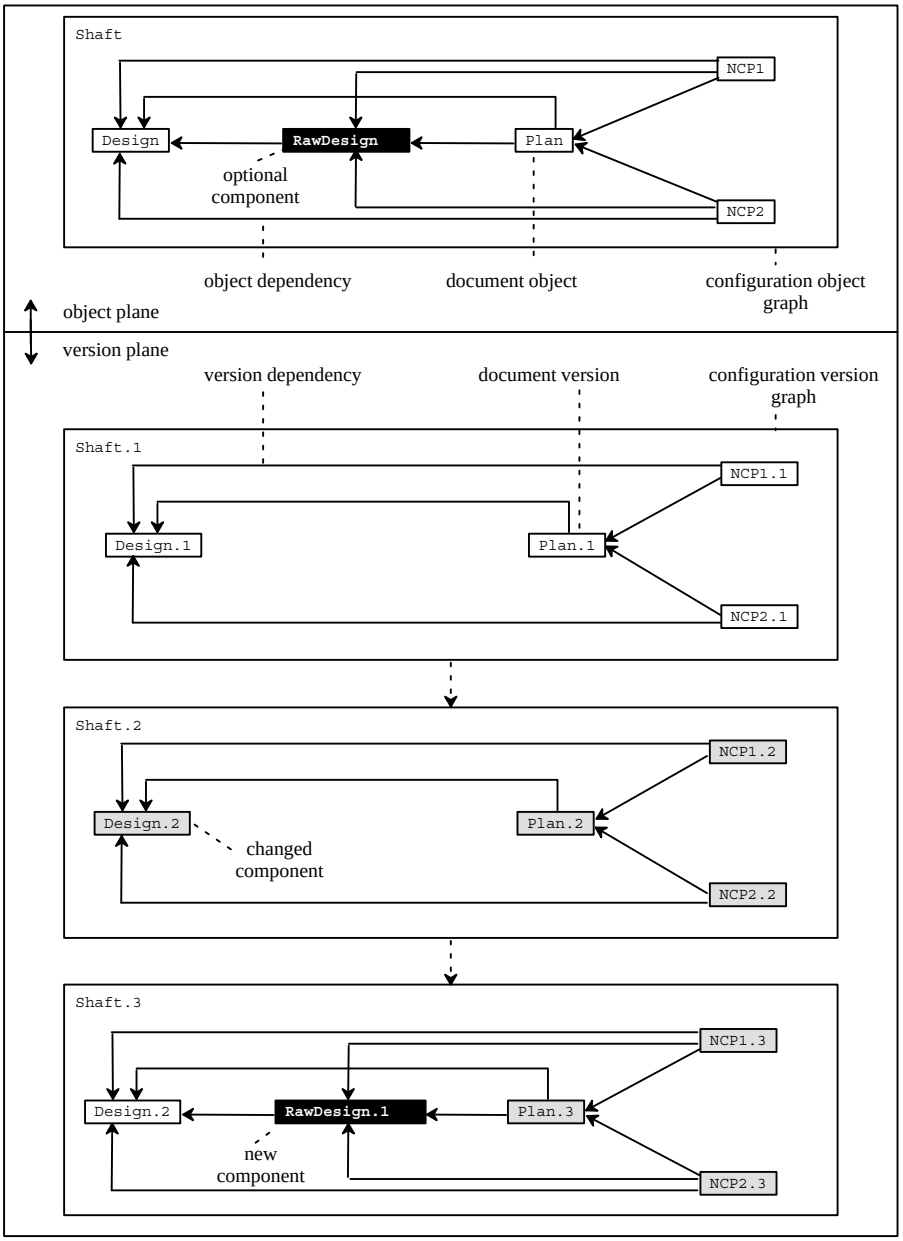


Figure 1.6: Configuration version graphs and configuration object graphs

mation in the object plane. The following constraint must hold between a configuration object graph and the corresponding configuration version graphs in the version plane: For each version component (dependency) contained in a configuration version graph, a corresponding object component (dependency) must exist in the configuration object graph. More precisely, each configuration version graph must be mapped by a graph monomorphism into the corresponding configuration object graph. This constraint guarantees that the version plane actually refines the object plane. Furthermore, an injective mapping excludes that multiple versions of the same object are contained in one configuration (version consistency). The upper part of Figure 1.6 shows a configuration object graph which satisfies these constraints.

Consistency control Product management assists in maintaining *consistency* between interdependent documents. Here, configuration versions play a crucial role. Instead of managing versions of documents individually and independently, configuration versions define sets of component versions among which consistency has to maintained. By recording configuration versions, consistent configurations may easily be reconstructed later on, e.g., when some bug needs to be fixed or a requirements change has to be realized. Guessing consistent combinations of component versions a posteriori is inherently difficult and error-prone.

More specifically, the product management model allows to distinguish among the following aspects of consistency:

- *Internal consistency* refers to the local consistency of a version. Internal consistency does not take relationships to other versions into account. For example, a version of an NC program is internally consistent if it consists of legal statements of the NC programming language.

- *External consistency* refers to the consistency of a dependent version with respect to a certain master. For example, a version of an NC program is consistent with a version of a CAD design if it conforms to the geometric data specified in the design.

- Finally, *component consistency* refers to the consistency of a version component with respect to a certain configuration version. Component consistency implies that the corresponding version is internally consistent, and that it is externally consistent with all master components.

1.3.2 Process Management Model

The process management model formalizes planning, analysis, control, and execution of development tasks. Process management is tightly integrated with product management. Furthermore, we put great emphasis on managing the dynamics of development processes.

Product-centered process management We follow a *product-centered approach* to process management. This means that we perform process management on top of product management by enriching the product structure with process data. This approach seems natural and is simple to understand. In particular, we avoid the conceptual overhead of keeping product and process structures consistent with each other (this has to be done if product and process are represented by separate structures).
In our product-centered approach, a configuration version 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 relationships 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 relationship starts at t_1 and ends at t_2, t_1 and t_2 are denoted as *predecessor* and *successor task*, respectively. Tasks can also be connected by (vertical) *composition relationships*: 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*. Finally, source and target of a composition relationship are called *supertask* and *subtask*, respectively.
An example of a task net is given in Figure 1.7[b]. 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).

Process dynamics It is widely recognized that development processes are highly *dynamic*. Due to their creative nature, it is rarely possible to plan a

[b] Note that flow relationships and dependencies have opposite directions.

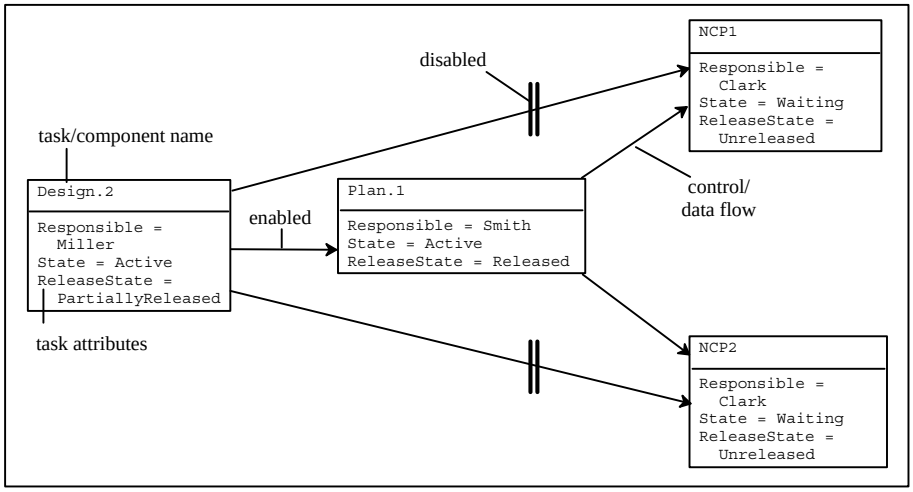


Figure 1.7: Example of a task net

development process completely in advance. Rather, many changes have to be taken into account during its execution. Some of them have been studied thoroughly in the SUKITS project, namely product evolution, feedback, and simultaneous engineering. These are discussed below.

Product evolution results in incremental extensions and structural changes of task nets. Let us consider again the task net of Figure 1.7. 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. Figure 1.7 shows a snapshot of the task net after the corresponding tasks have been inserted. Later on, the manufacturing plan may still be changed, resulting in a different set of NC programs and corresponding modifications to the task net.

The "classical", conservative rule for defining the execution order states that a task can be started only after all of its predecessors have been finished. However, enforcing this rule significantly impedes parallelism. In order to shorten development time, the conservative rule needs to be relaxed so that tasks may be executed concurrently even if they are connected by flow relationships. *Simultaneous engineering* [35] denotes development methods to achieve this.

Simultaneous engineering 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 feedback from later phases in the life cycle into account (e.g., design for manufacturing).

Versions may be prereleased *selectively* to successor tasks. An example is given in Figure 1.7, where the design task has the release state `PartiallyReleased`. Attributes attached to flow relationships control which successors may access a selectively released version. In Figure 1.7, 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.

Due to simultaneous engineering, 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 1.8. 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.

Start and termination of tasks are controlled by activation and termination conditions, respectively. The *activation condition* requires that all essential inputs are available, i.e., they are (pre-)released by predecessor tasks. "Essential" weakens the activation condition by enforcing availability of only those inputs which are needed to start working. On the other hand, the *termination condition* is much more restrictive and conservative. All predecessor tasks must have terminated and released their output to the successor task; furthermore, all inputs must be up to date, i.e., the final version of each input document

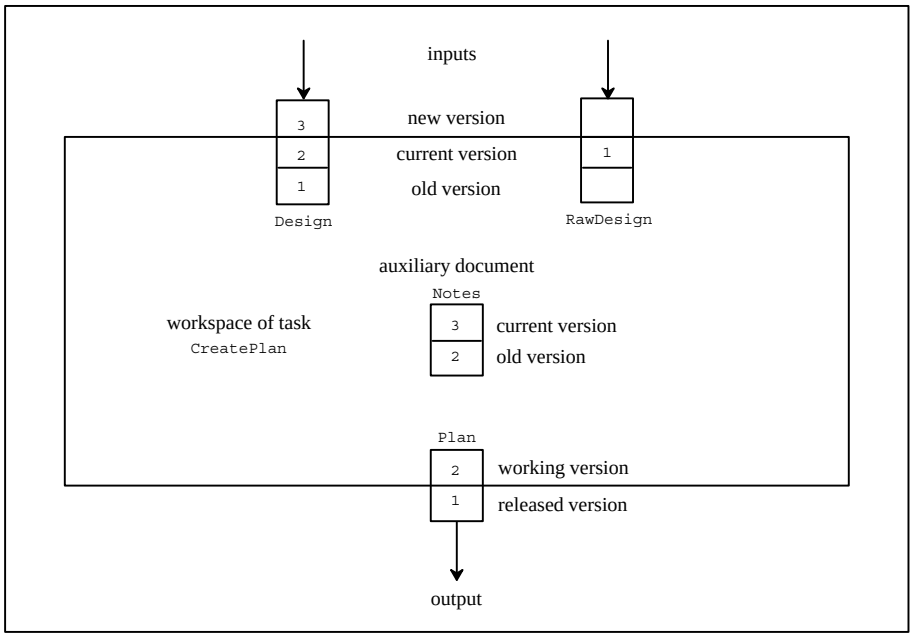


Figure 1.8: Workspace of a task

must have been consumed. These restrictions prevent premature termination, which could require reactivation of a task when its inputs are changed later on. Note that the conservative termination condition does not slow down the work; rather, it alerts the engineer of possibly changing inputs and merely delays a state transition.

However, reactivation of terminated tasks cannot be excluded completely because of *feedback* occurring in the development process. Feedback is raised by a task t_1 which detects some problem regarding its inputs. This problem is caused by a task t_2 which precedes t_1 in the task net. In general, t_2 may be a transitive predecessor of t_1. If t_2 has already terminated, it has to be reactivated.

The impact of feedback may be hard to predict, and it may vary considerably from case to case. t_2 may raise cascading feedback to a predecessor t_3. Otherwise, t_2 accepts the responsibility for processing feedback. Then, the impacts on transitive successors of t_2 need to be estimated. Active tasks which will be affected drastically have to be suspended even if they are not immediate successors of t_2. Other tasks will not be affected at all and may continue with-

out disruption. Thus, feedback needs to be managed in a flexible way, relying heavily on decisions to be performed by the user. In general, it is virtually impossible to automate the processing of feedback completely, e.g., by brute-force methods such as suspending all transitive successors.
Product evolution, simultaneous engineering, and feedback are illustrated in Figures 1.9 and 1.10, which show an evolving task net for the development of an assembly part (a gear). Task states are represented by different symbols or fill patterns (e.g., a grey box highlights an active task). The figure shows five snapshots of the task net, corresponding to different stages of development:

1. After having defined the requirements, 3-D model and part list are created in parallel (simultaneous engineering). Since the product structure has not been fixed yet, the rest of the task net is still unknown.

2. Both the 3-D model and the part list are available; the corresponding tasks have terminated. The product structure is defined by the entries contained in the part list. In the following, we assume that the gear consists of two gear wheels and one shaft. For each component of the gear, corresponding tasks for creating the drawing and the manufacturing plan are inserted into the task net. These insertions need not be performed by hand; rather, a tool may be used which extracts the product structure from the part list and extends the task net accordingly. NC programming tasks cannot be created yet because the NC programs to be prepared are defined only in the manufacturing plan.

3. The development process has advanced further with respect to the single parts the gear is composed of. In the case of the shaft, both design and manufacturing planning are being carried out in parallel. In the case of the first gear wheel, design and manufacturing planning have already completed, and the NC programs for the steps defined in the manufacturing plan are being prepared. In the case of the second gear wheel, the manufacturing steps have been fixed, but the drawing is not yet finished. Therefore, NC programming cannot be started yet.

4. The customer changes the requirements to the gear (externally induced feedback). Requirements definition and 3-D model need to be adapted; therefore, the corresponding tasks are activated again. However, further consequences of the requirements change cannot be determined yet. Rather, we have to wait for more detailed information to be extracted from the revised design. Several tasks which have already been completed may be affected by the change (creation of the part list, design of and

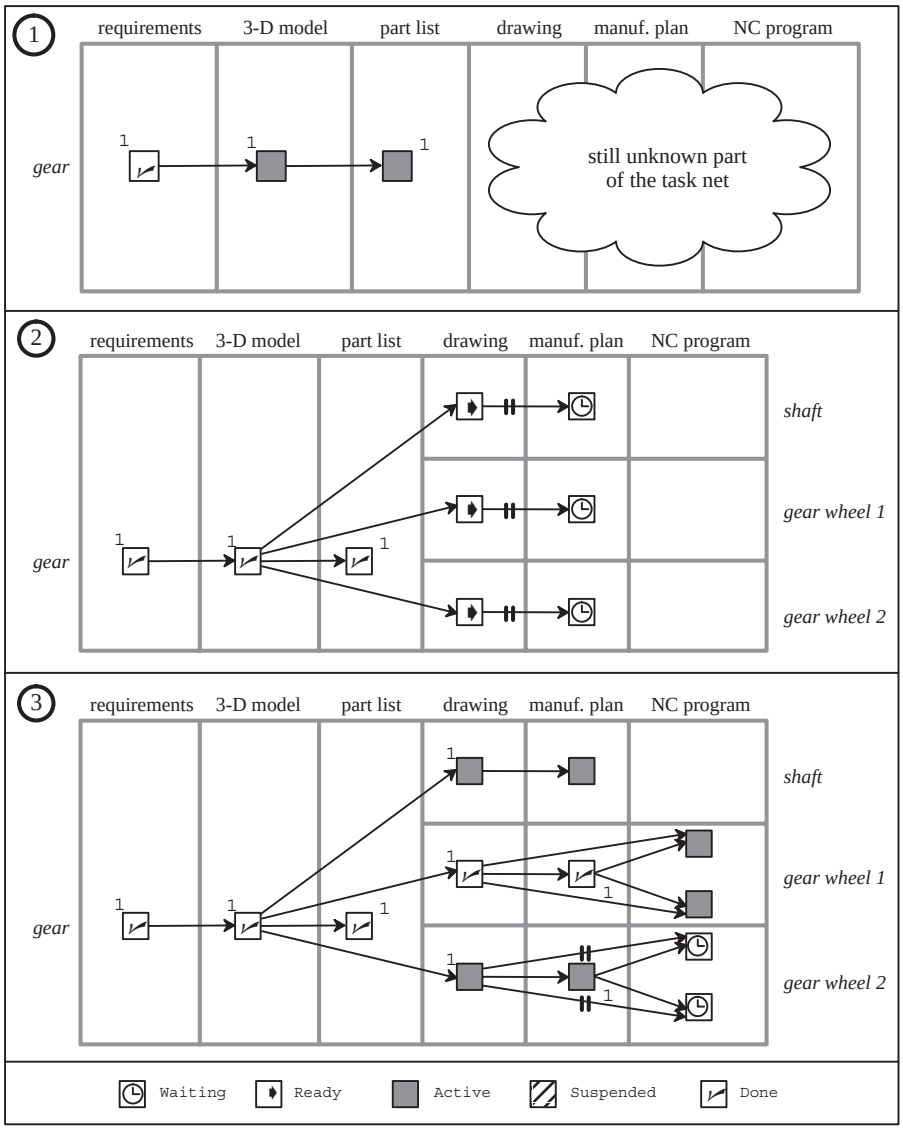


Figure 1.9: Example for evolving task nets

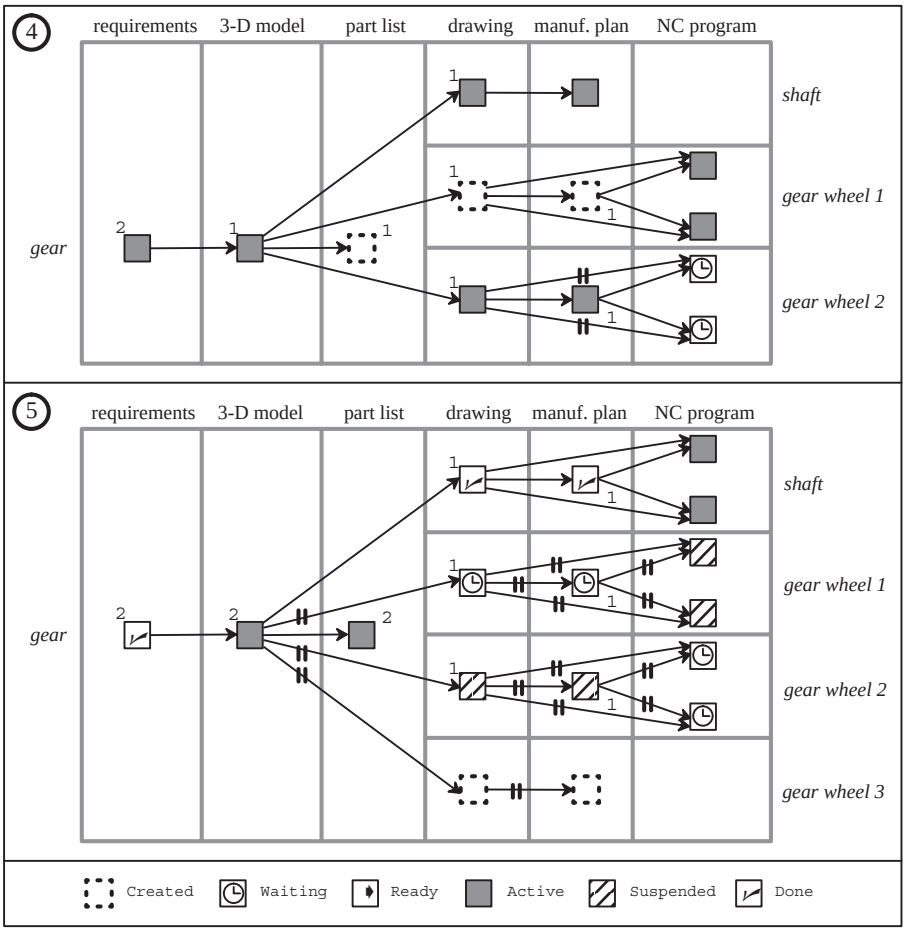


Figure 1.10: Example for evolving task nets (continued)

manufacturing planning for the first gear). These tasks are forced to return to the state `Created`. Later on, they are either restarted, or their previous results are reused without any modification.

5. In order to meet the changed requirements, the designer of the gear decides to add another gear wheel. This modification also affects the existing gear wheels because their transmission ratios need to be changed. Now the impact of the change can be determined more accurately. The shaft, whose drawing and manufacturing plan have been created in the meantime, remains unaffected. Development of gear wheels 1 and 2 is suspended temporarily until the details of the design change are known. Note that releases have to be revoked both selectively (successors of the design task for the assembly part) and transitively (suspension of all tasks for gear wheels 1 and 2). For the third gear wheel, another branch is added to the task net.

1.4 Management Model: Formal Specification

To formalize the management model which was described informally in the last chapter, we use the specification language *PROGRES*. Note that the specification given below concerns the *internal representation* used by management tools. In contrast, in the previous section several figures showed *external representations* of graphs visible at the user interface. These levels have to be separated clearly. Internal representations include all details which have to be taken care of by sophisticated management tools. On the other hand, a simple and readable representation is required at the user interface.
In the following, we present cut-outs of the specification for the management of products and processes. The specification fragments to be presented were created with the help of the PROGRES environment. In our representation, we do not assume that the reader is familiar with the PROGRES language. Rather, the language constructs are explained as required. For further information on PROGRES, the reader is referred to [36] and Chapter **??** of this volume.

1.4.1 Product Management Model

Graph Schema PROGRES is based on *attributed graphs*. An attributed graph consists of nodes decorated with attributes and edges which do not carry attributes. An edge represents a binary, directed relationship. When a node is deleted, all adjacent edges are deleted as well (referential integrity). Furthermore, an edge may be traversed in both directions (from source to target and vice versa).

As in database management systems, the elements of graphs of a certain class — e.g., product management graphs to be discussed below — are declared in a *graph schema*. Nodes and edges are classified into node classes and edge types, respectively. A *node class* defines the attributes which its instances will carry. An *edge type* defines the classes of source and target nodes and the cardinality of its instances.
Node classes are organized into a *multiple inheritance hierarchy*. A subclass inherits from its superclasses all attributes and adjacent edge types. Inheritance on edge types is not supported.
A graph schema may be represented both textually and graphically. Figure 1.11 displays a graphical *schema diagram* for product management graphs; later on, we will also use the textual notation as required. In the diagram, each node class is represented by a box which contains the name of the class and its attributes. Dashed arrows stand for inheritance relationships (`is_a`). Edge types are represented by solid arrows which are decorated with type names and cardinalities. For example, the edge type `OC_Contains` connects `1` node of class `CONFIGURATION_OBJECT` to `n` nodes of class `OBJECT_COMPONENT` (i.e., `OC_Contains` is a `1:n` relationship).
The graph schema was designed according to the following rules:

- Relationships, which were represented as arrows in the figures presented in Section 1.3.1, are modeled as nodes and adjacent edges (e.g., a successor relationship is modeled as a `HISTORY` node and `Predecessor/Successor` edges). This solution allows for attaching attributes to relationships, and for establishing relationships between relationships.

- Each subgraph of the product management graph is represented by a root node which is connected to all nodes belonging to this subgraph (e.g., a configuration version graph is represented by a `CONFIGURATION_VERSION` node which is connected to `VERSION_COMPONENT` and `VERSION_DEPENDENCY` nodes by `VC_Contains` and `VD_Contains` edges, respectively. The graph model is constructed such that subgraphs are mutually disjoint. In particular, we distinguish between applied occurrences (e.g., `OBJECT_COMPONENT`) and declarations (e.g., `OBJECT`).

- Apart from `HISTORY` nodes, each node of the version plane is connected to the corresponding node of the object plane by an (incoming) `HasInstance` edge. Such a node of the version plane may be regarded as an instance of exactly one node of the object plane. In this way, the

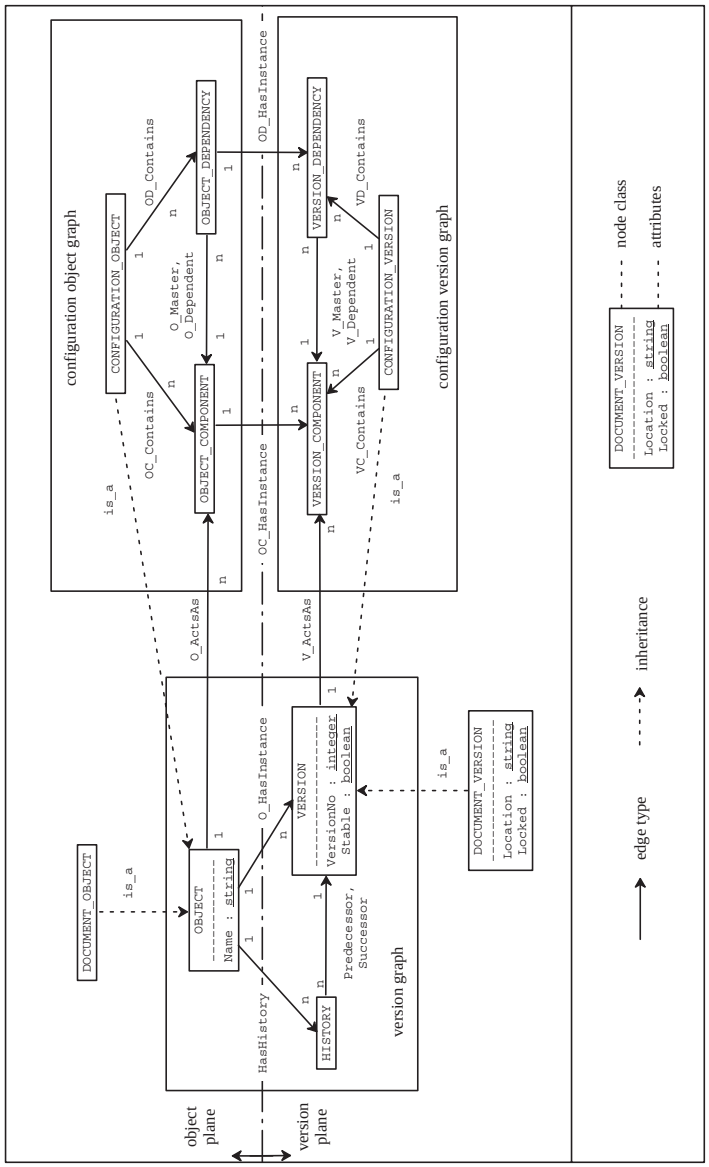


Figure 1.11: Graph schema for product management

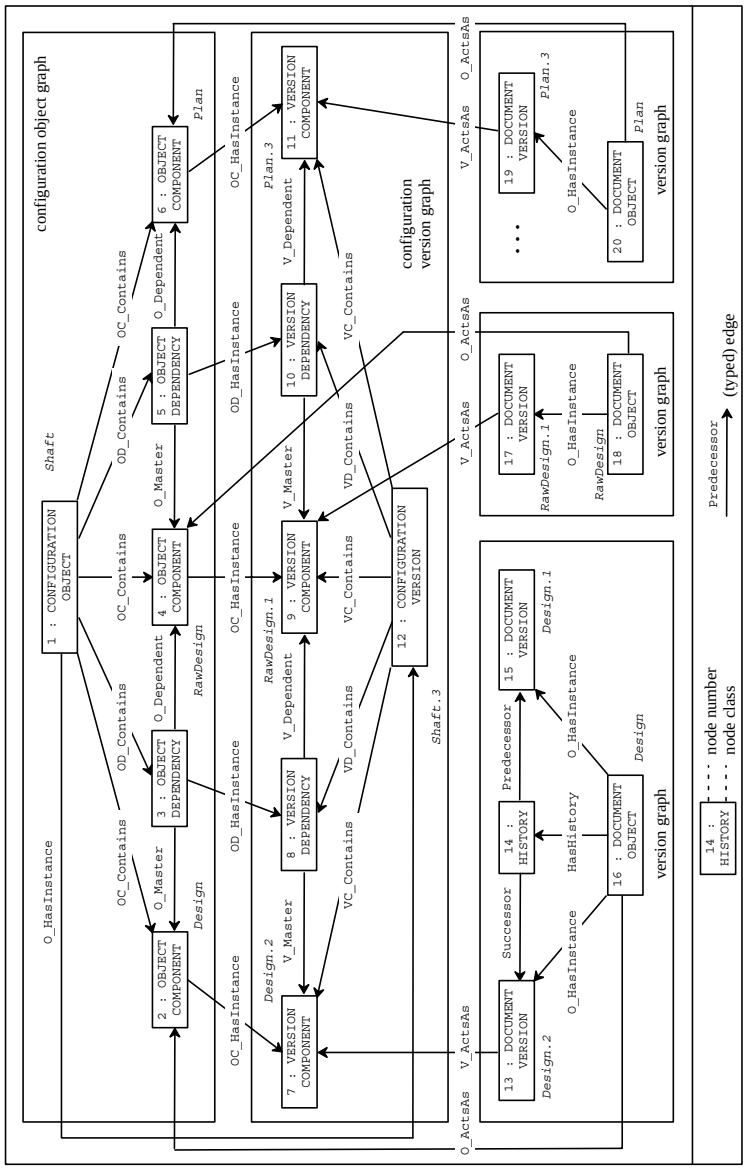


Figure 1.12: Example of a product management graph

refinement relationship between object plane and version plane is represented. Note that this kind of instantiation is user-defined (on top of PROGRES) and has to be distinguished from PROGRES instantiations (e.g., a node is an instance of a node class).

More specifically, the graph schema is structured as follows:

- Each `OBJECT` carries a unique `Name` attribute. The corresponding version graph contains `VERSION` and `HISTORY` nodes. `VersionNo` identifies a version uniquely among the versions of one object. `Stable` indicates whether the version may still be modified. Furthermore, a `DOCUMENT_VERSION` carries additional attributes `Location`, which identifies the place where the version contents are stored, and `Locked`, which indicates whether the version has been reserved for modification.

- A `CONFIGURATION_OBJECT` graph contains `OBJECT_COMPONENT` and `OBJECT_DEPENDENCY` nodes. Each `OBJECT_COMPONENT` is connected to its corresponding `OBJECT` by an incoming `O_ActsAs` edge.

- A `CONFIGURATION_VERSION` graph contains nodes of class `VERSION_COMPONENT`, which are connected to the corresponding `VERSION` and `OBJECT_COMPONENT` by an incoming `V_ActsAs` and `OC_HasInstance` edge, respectively, and nodes of class `VERSION_DEPENDENCY`, each of which has an incoming `OD_HasInstance` edge from the corresponding `OBJECT_DEPENDENCY`.

Figure 1.12 shows an example of a product management graph which is derived from the external representation in Figure 1.6 on page 11. The configuration object graph for `Shaft` is displayed on the top. Below, one of its versions is shown, namely `Shaft.3`. At the bottom, version graphs for the documents `Design`, `RawDesign`, and `Plan` are given (see Figure 1.5 on page 10 for the external representation of version graphs). To avoid cluttering of the figure, we show only cut-outs of the different graph structures (the version graphs are not complete, and the dependencies between `Plan` and `Design` were omitted).

Constraints So far, the graph schema is not powerful enough to express all kinds of *constraints*. Further structural constraints are given in Table 1.1. These constraints are classified by attributes (third and fourth column) which are discussed below.
An *enforced* constraint must hold after each graph transformation. More precisely, considering a graph class an abstract datatype, we require that an en-

no	*name*	*enforced*	*static/ dynamic*	*description*
1	Unique naming	yes	static	Object names must be globally unique.
2	Unique version numbers	yes	static	Versions of one object must be numbered in a unique way.
3	Unique object components	yes	static	An object must not act more than once as a component in the same configuration object.
4	Unique relations	yes	static	For each pair of versions (components), there must exist at most one connecting successor (dependency) relationship.
5	Local relations	yes	static	Successor and dependency relationships must be local, i.e., both ends must belong to the same subgraph.
6	Acyclic relations	yes	static	Cycles in successor and dependency relationships are not allowed.
7	Object–version correspondence	yes	static	Each version component (dependency) has to be mapped 'monomorphically' to the corresponding object component (dependency).
8	Version deletion condition	yes	dynamic	A version which has applied occurrences must not be deleted.
9	Locking	yes	dynamic	A document version can only be checked out once for writing.
10	Check in	yes	dynamic	A document version can only be checked in if it has been checked out for writing.
11	Internal consistency	no	static	The contents of each version must eventually be consistent. For configuration versions, this means that each component must eventually be consistent.
12	External consistency	no	static	For each version dependency, the dependent component must eventually be consistent with the master component.
13	Component consistency	no	static	Each version component must eventually be bound to an internally consistent version, and it must eventually be externally consistent with all master components.
14	Consistency of stable versions	yes	static	Each stable version must be internally consistent.
15	Historical stability	yes	static	Each version which has a successor must be stable.
16	Configuration stability	yes	static	All components of a stable configuration version must be stable, as well.
17	Change protection	yes	dynamic	A stable configuration version must not be modified.

Table 1.1: Consistency constraints for product management

```
node class VERSION
  constraint
    InternallyConsistent = true;
end;

node class DOCUMENT_VERSION is a VERSION
  redef constraint
    InternallyConsistent = CheckConsistency ( self.Location );
end;

node class CONFIGURATION_VERSION is a VERSION
  redef constraint
    InternallyConsistent =
      for all Component : VERSION_COMPONENT := self.-VC_Contains-> ::
        Component.ComponentConsistent
      end;
end;

node class VERSION_COMPONENT is a NODE
  constraint
    ComponentConsistent =
      [ self is with <-V_ActsAs- ::
        use Version : VERSION := (self.<-V_ActsAs- : VERSION [1:1]) ::
          Version.InternallyConsistent
        end
        and
        for all
          IncomingDependency : VERSION_DEPENDENCY := self.<-V_Master- ::
          IncomingDependency.ExternallyConsistent
        end
      | false ];
end;
```

Figure 1.13: Definition of textual constraints

forced constraint be fulfilled after each execution of an exported graph transformation. Whether the graph transformation is atomic (a production) or complex (a transaction), does not matter.

As far as possible, constraints are enforced. Only a few constraints can be violated temporarily (internal, external, and component consistency, respectively; see Conditions 11–13). These constraints are enforced when a version is frozen (14). Thus a version may be inconsistent as long as it is being modified (otherwise, the user were forced to maintain consistency even in early phases of development, which is clearly unrealistic).

A constraint is either static or dynamic. A constraint is called *static* if it defines an invariant. These invariants must be preserved by all graph transformations. More precisely, all exported graph transformations violating enforced invariants will fail; invariants which are not enforced will have to hold eventually. In Table 1.1, most constraints are static, e.g. "Configuration stability" (16).

In some cases, static constraints alone are not sufficient. For example, "Config-

uration stability" alone does not guarantee that a stable configuration version can no longer be modified (insertion/deletion of components/dependencies). A *dynamic* constraint restricts the applicability of graph transformations to certain graph states. For example, "Change protection" (17) excludes changes to stable configuration versions.

Constraints can be expressed in PROGRES in several ways. Dynamic constraints are formalized by *pre-* and *postconditions* which are given either explicitly or implicitly as left-hand and right-hand sides of graph rewrite rules. Static constraints can be expressed in the graph schema by *attribute constraints*. An attribute constraint is a *derived attribute* which is defined by a boolean expression. In general, PROGRES distinguishes between intrinsic attributes, which receive values through explicit assignment, and derived attributes, which are calculated automatically from (intrinsic or derived) attributes in the neighborhood.

Constraints can be defined either textually or graphically. Examples of *textual constraints* are given in Figure 1.13:

1. In the class `VERSION`, a constraint `InternallyConsistent` is introduced. Since no meaningful evaluation rule can be defined at this general level, the constraint is constantly `true`.

2. In the subclass `DOCUMENT_VERSION`, the constraint is redefined. The function `CheckConsistency` is used to check the internal consistency of the contents. Recall that `Location` refers to the address (e.g., file path name) under which the contents are stored.

3. Internal consistency is defined differently for configuration versions: All component versions must be consistent. `for all` denotes universal quantification over all elements in a set (the components of the configuration, which are reached from the current node `self` via outgoing `VC_Contains` edges).

4. The last condition expresses "Component consistency" (Constraint 13 in Table 1.1). At the top level, the evaluation rule consists of a list of subexpressions. In this case, the list is a pair $[e_1 \mid e_2]$. e_1 is a guarded expression of the form $c_1 :: b_1$. c_1 checks whether the current node has an incoming `V_ActsAs` edge. If this is not the case, the version component is still *unbound* (no referenced version). Then, the guard fails, and e_2 is selected, i.e., the component is considered (temporarily) inconsistent. Otherwise, the component is *bound* to a specific version, resulting in success of the guard. Then, b_1 is selected for evaluation. b_1 checks whether

the corresponding version is internally consistent (note that `use` introduces a local variable). Furthermore, for each outgoing dependency it is checked whether the component is externally consistent with its master (the constraint `ExternallyConsistent` is attached to dependencies).

Figure 1.14 shows an example of a *graphical constraint*. A successor relationship is consistent if it meets the graphical restriction `HistoryRestriction`. In general, a *restriction* is a predicate defined on nodes of a certain class (here: `HISTORY`). A graphical restriction is fulfilled if the corresponding graph pattern can be matched in the host graph. There is one designated node of the pattern, namely the node for which the restriction is to be checked (node `‘3`, specified above the graph pattern). The graph pattern ensures that source (`‘2`) and target (`‘4`) belong to the same version graph (locality, Constraint 5). Furthermore, the negative node `‘5` expresses a negative application condition which excludes duplicates of successor relationships (4). Finally, further constraints are checked in the textual condition part below the graph pattern. The first condition excludes cycles: Since the predecessor must have a smaller version

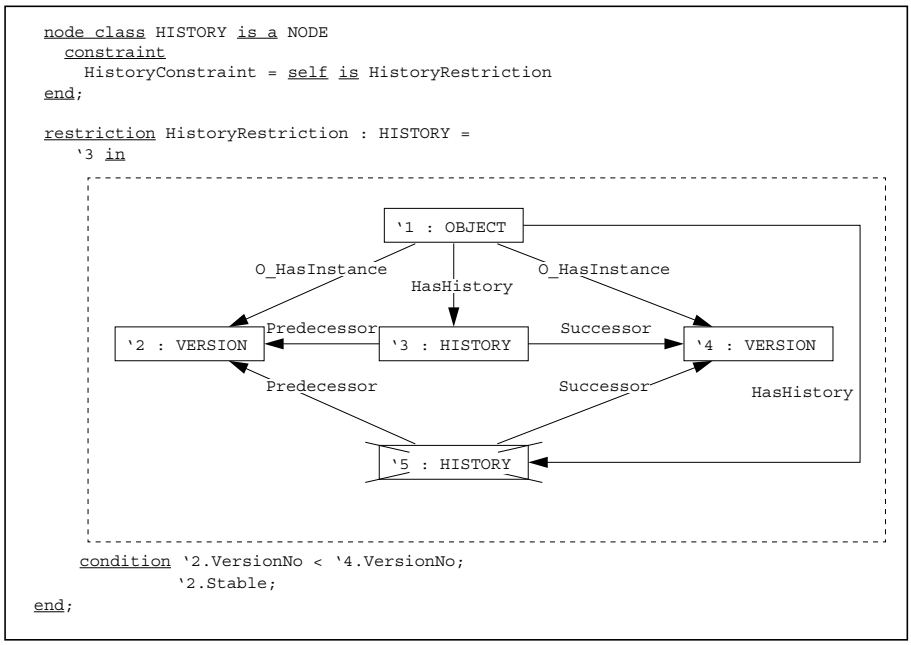


Figure 1.14: Definition of a graphical constraint

number than the successor, cycles cannot occur (6). The second condition ensures historical stability (15): Each version (`‘2`) which has a successor must be stable.
Note that the definition of constraints is given as part of the graph schema. Comparing textual and graphical notations used for expressing the graph schema, we observe that the textual notation is sufficient in principle: the graphical notation can always be transformed into an equivalent textual notation. On the other hand, the graphical notation is more illustrative. Therefore, both kinds of notation are offered to the PROGRES user, who can select the most convenient notation pragmatically.

Graph transformations PROGRES provides *graph rewrite rules* for specifying graph transformations in a declarative and graphical way. “Declarative” means that the user is not concerned with the algorithm that is executed in order to apply a graph rewrite rule. In particular, the user need not worry about how an instance of the left-hand side is searched in the host graph (once found, replacement with the right-hand side is rather straightforward). All graph transformations are checked for consistency with the graph schema.
A graph rewrite rule (also called *production*) consists of the following parts: The header is composed of an identifier and a list of formal parameters. The left-hand side describes the subgraph to be replaced. The right-hand side specifies the subgraph to be inserted. The condition part lists conditions on attributes of nodes belonging to the left-hand side. The transfer part assigns values to attributes of nodes belonging to the right-hand side. Finally, result parameters receive values in the return part. Left-hand side and right-hand side are mandatory, all other parts are optional.
For each kind of subgraph, we will give just one example of a graph rewrite rule. Rules for version graphs and configuration version graphs are described below; a rule for configuration object graphs will follow in Section 1.4.3.
Figure 1.15 shows a graph rewrite rule for inserting a successor relationship into a *version graph*:

1. The production `CreateHistory` receives the `PredecessorVersion` and the `SuccessorVersion` as input parameters and returns the `NewHistory` relationship as output.

2. On the left-hand side, the nodes `‘2` and `‘3` are fixed by the input parameters `PredecessorVersion` and `SuccessorVersion`, respectively. This is indicated by the notation n=p, where n and p denote a node of the left-hand side and an input parameter, respectively. Furthermore, node `‘4`,

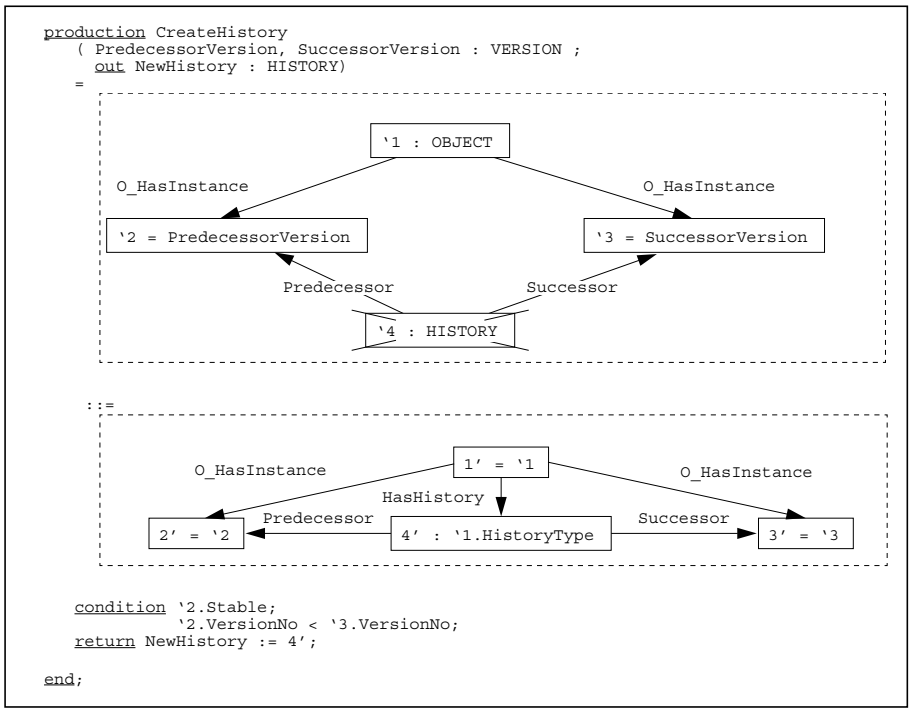


Figure 1.15: Creation of a successor relationship

which is crossed through, represents a negative application condition. Finally, the double arrows emanating from node `1 denote paths (derived binary relationships, see below).

3. The condition part, which is given below the right-hand side, supplements the left-hand side by defining conditions on attribute values (referring to the `Stable` attribute of node `2 and the `VersionNo` attributes of nodes `2 and `3).

4. The right-hand side, which appears below the left-hand side, describes the replacing subgraph. In this example, all nodes occurring on the left-hand side as well as their connecting edges are replaced identically, which is indicated by the notation r=l (where l and r denote nodes of the left-hand side and right-hand side, respectively). Furthermore, a new node `4'` is created and connected to its neighbors[c].

[c] For now, let us ignore the meaning of the expression following the colon. It is sufficient

5. In this sample rule, the transfer part is empty.

6. Finally, the return part (at the bottom) assigns the new node `4'` to the output parameter `History`.

The alert reader will have noticed the close correspondence between the graph rewrite rule of Figure 1.15 and the restriction of Figure 1.14. In fact, we have described in [15] how graph rewrite rules can be derived mechanically — yet manually — from graphical constraints. Thus, constraints provide considerable assistance in specifying consistency-preserving operations.
An example of a graph rewrite rule modifying a *configuration version graph* is given in Figure 1.16. `CreateVersionDependency` receives the components to be connected as input parameters. The right-hand side inserts a dependency node and connects it to the master and the dependent component, respectively, to the configuration version containing them, and to the matching object dependency. The left-hand side checks a lot of constraints: The components must belong to the same configuration version (`'7`), a dependency must not yet exist (negative path from `'6` to `'5`; explanation see below), and there must be a matching object dependency (`'4`)[d]. Furthermore, the condition part checks that the configuration version is not frozen (Constraint 17).
Duplication of dependencies is excluded by a negative application condition which makes use of a *path*. In general, a *path* is a binary, derived relationship which is defined by a path expression. A path expression can be written in a graphical way (similarly to a graphical restriction, see e.g. Figure 1.14) or textually. In the latter case, paths can be composed by operators such as sequence (p_1 `&` p_2), transitive closure (p`+`), alternative (p_1 <u>`or`</u> p_2), etc. For example, the path `DependsOnVersion` is defined as `<-V_Dependent- & -V_Master`, i.e., as a concatenation of two edge traversals (the first one in negative and the second one in positive direction).

1.4.2 Process Management Model

According to our product-centered approach, we define the process management model as an "extension" to the product management model. Conceptually, this means that we apply *inheritance* at the level of *graph classes*, i.e., the graph class "process management graph" inherits from the graph class "product management graph". Currently, PROGRES only supports inheritance at the level of node classes (actually, the notion of graph class is still implicit in the

to understand that a node of class `HISTORY` is created; further explanations will follow in Section 1.4.3.

[d] Node `'1` is redundant and is only included for the sake of readability.

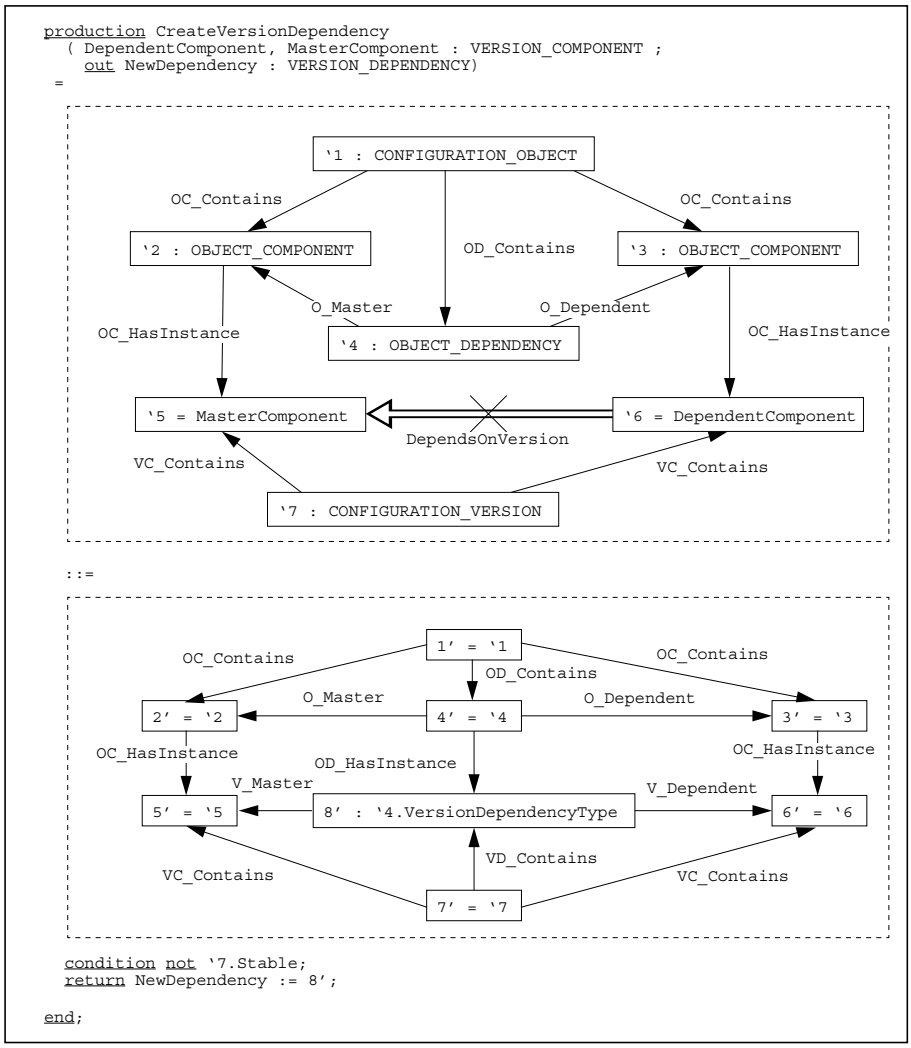


Figure 1.16: Creation of a version dependency

```
node class TASK is a VERSION_COMPONENT
  intrinsic
    State : TASK_STATE := Created;
    ReleaseState : RELEASE_STATE := Unreleased;
end;

edge type WorksOn : TASK [0:1] -> VERSION [0:1];

node class DATA_FLOW is a VERSION_DEPENDENCY
  intrinsic
    Propagate : boolean;
    NeededForActivation : boolean := false;
    CurrentValid : boolean := false;
    OldValid : boolean := false;
end;

edge type Current : DATA_FLOW [0:n] -> VERSION [0:1];

edge type Old : DATA_FLOW [0:n] -> VERSION [0:1];
```

Figure 1.17: Graph schema for product-centered process management

language). Below, inheritance between graph classes is expressed by extending the graph schema, redefining operations and introducing new operations.

Graph schema The graph schema for task nets is defined by refining the schema for configuration versions introduced in Section 1.4.1. In Figure 1.17, the schema extensions are described in a textual notation (recall that a graph schema can be written both textually and graphically). The keyword `intrinsic` indicates *intrinsic attributes*, which are explicitly assigned values (in contrast to derived attributes). An *edge type declaration* specifies the classes of source and target nodes as well as permitted cardinalities (see below).

`TASK` is a subclass of `VERSION_COMPONENT` to which a `State` attribute is attached (to be discussed below). Furthermore, the `ReleaseState` indicates whether the task output is `Unreleased`, `PartiallyReleased` to some successors, or totally `Released`. An edge of type `WorksOn` points from a task to its working version. Each task may have at most one working version (cardinality `[0:1]` following the target class specification). Conversely, it is not allowed that multiple tasks share the same working version (cardinality `[0:1]` following the source class specification). The released version is represented by an incoming `V_ActsAs` edge, as defined in the product management schema (see Figure 1.11 on page 22).

Analogously, `DATA_FLOW` is defined as a subclass of `VERSION_DEPENDENCY`. In case of selective releases, the `Propagate` attribute controls whether the output of the preceding task may be transmitted along the data flow. `NeededForActivation` describes whether an input must be available in order to start execution. Edges of type `Current` and `Old` point to the currently and pre-

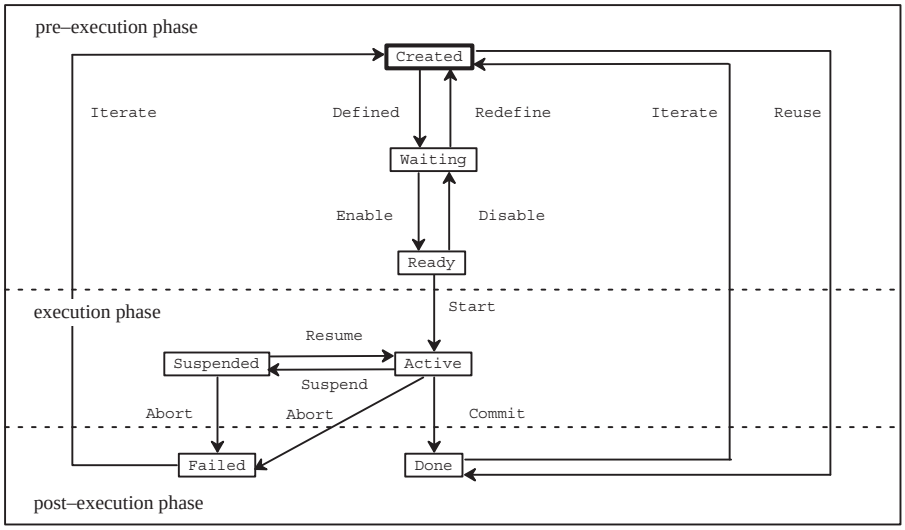


Figure 1.18: State transition diagram for tasks

viously consumed input version, respectively. Accordingly, the attributes `CurrentValid` and `OldValid` describe whether the current and the old version may be used by the task located at the target of the data flow, respectively.

Constraints We have already demonstrated in the previous subsection how constraints can be described formally and informally. In the sequel, constraints are defined with the help of semi-formal notations which are well suited for process management. Most of them are dynamic constraints, which have to be checked in graph transformations anyway. Some constraints are static (the compatibility matrices for states of neighboring tasks), but we refrain from demonstrating how they are encoded as PROGRES constraints (see previous section). In all cases, the constraints are enforced, i.e., they must hold after each exported operation.

The *state transition diagram* in Figure 1.18 defines in which states a task may reside and which operations affect state changes. `Created` serves as initial state which is also restored in case of feedback. The `Defined` transition indicates that the task definition has been completed. In case of feedback, `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`. As soon as the activation condition is fulfilled, the transition `Enable` moves the task into the

state `Ready`. Subsequent violation of the activation condition results in the inverse transition `Disable` back to `Waiting`. `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.

supertask ▶ *subtask* ▼	Created	Waiting	Active	Suspended	Done	Failed
Created	+	+	+	+	–	+
Waiting	+	+	+	+	–	+
Active	–	–	+	–	–	–
Suspended	–	–	+	+	–	–
Done	+	+	+	+	+	+
Failed	+	+	+	+	–	+

Table 1.2: Compatibility matrix for task states (vertical relationships)

pred. task ▶ *succ. task* ▼	Created	Waiting	Active	Suspended	Done	Failed
Created	+	+	+	+	+	+
Waiting	+	+	+	+	+	+
Active	+	+	+	+	+	+
Suspended	+	+	+	+	+	+
Done	–	–	–	–	+	–
Failed	+	+	+	+	+	+

Table 1.3: Compatibility matrix for task states (horizontal relationships)

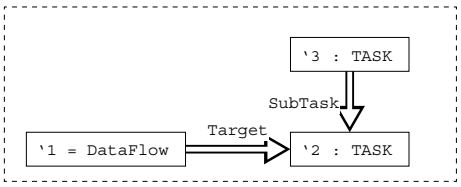
state ▶ *operations* ▼	Created	Waiting	Active	Suspended	Done	Failed
structural changes: task interface	+	–	+	+	–	–
structural changes: task realization	–	–	+	–	–	–
workspace management	–	–	+	–	–	–
release management	+	–	+	+	–	–

Table 1.4: Legal operations in different task states

Tables 1.2–1.4 supplement the description of the dynamic behavior of tasks. In Table 1.2, legal combinations of tasks connected by *vertical* (composition) *relationships* are specified by a *compatibility matrix*. Since `Created` and `Waiting` belong to the pre-execution phase (see Figure 1.18), subtasks residing in the execution phase (`Active` or `Suspended`) are not allowed. In case of an `Active` supertask, subtasks may be in any state. A `Suspended` supertask must not have an `Active` subtask: If a supertask is suspended, all subtasks are suspended as well. A supertask can only be `Done` if all subtasks have committed. Finally, no subtask of a `Failed` supertask may be `Active` or `Suspended`.
Similarly, the compatibility matrix in Table 1.3 refers to tasks which are connected by *horizontal relationships* (data/control flows). The only constraint is that all predecessors of a `Done` task must be `Done` as well (conservative termination condition).
Finally, Table 1.4 indicates in which states certain classes of operations are allowed and prohibited, respectively. In state `Created`, it is only possible to define the interface of a task (inputs to be consumed and outputs to be produced). The realization of a complex task (the refining task net) is built up on the fly by the responsible manager after execution has started. Neither may inputs be consumed nor outputs be produced (workspace management); these operations belong to the execution phase. Finally, release states can be changed in order to handle feedback. `Waiting`, `Done`, and `Failed` are "dead" states in which no operations may be carried out. In state `Active`, all classes of operations may be applied, resulting in seamless integration of editing, analysis, and execution. Finally, in state `Suspended` only the task interface and the release information may be modified. In order to consume inputs/produce outputs, or to define the realization of a complex task, execution needs to be resumed.
Note that the state transition diagram and the tables given above define the constraints which have to be regarded by graph transformations only partially. In addition, further constraints on data flows must be taken into account. In particular, a task may be started only when all essential inputs are available. In case this condition is violated after activation, execution must be suspended. Furthermore, a task may only commit if all inputs are up to date (i.e., the task has seen the last value of each input) and the output has been released totally.

Graph transformations The graph transformations for process management must respect the constraints given by the state transition diagram and the compatibility matrices introduced above. Again, the graph transformations can be derived from the constraints in a mechanical way. For example, a graph transformation for a state transition must check the source state of the corresponding task and update the state attribute such that it holds the

```
transaction DeleteDataFlow( DataFlow : DATA_FLOW) =
    CheckFlowDeletionConstraints ( DataFlow )
  & DeleteVersionDependency ( DataFlow )
end;

test CheckFlowDeletionConstraints( DataFlow : DATA_FLOW) =
```




```
  condition '3.State = Active;
            '2.State in (Created or Active or Suspended);
end;
```

Figure 1.19: Deletion of a data flow

target state after the graph transformation. Further application conditions are derived from the compatibility matrices.

Graph transformations for process management may be classified into four categories: structural changes, state transitions, workspace management, and release management. For each of these categories, one example is given below.

Graph transformations for *structural changes* are obtained by redefining the corresponding operations introduced in the product management model. These operations are checked against the constraints of the process management model.

To redefine product management operations, we use *transactions* for programming with graph rewrite rules. PROGRES provides several control structures for composing graph rewrite rules, including sequence, various forms of loops and branches, etc. The term "transaction" was borrowed from database management systems to indicate atomicity, consistency, and durability (isolation is not mentioned here because PROGRES does not deal with concurrency control; this is handled by the underlying database management system GRAS [37]). In particular, in case of failure all changes are undone. For example, if the rule r_2 cannot be applied after r_1, a transaction which calls r_1 and r_2 sequentially fails, and the effects of r_1 are removed.

For example, `DeleteDataFlow` (Figure 1.19) performs a test and then applies `DeleteVersionDependency` (operator `&`). `CheckFlowDeletionConstraints` ensures that the constraints of Table 1.4 are not violated. First, a task net may only be modified if its root (node `'3`) is `Active`. Second, the state of the target task (node `'2`) must allow for changes to its interface.

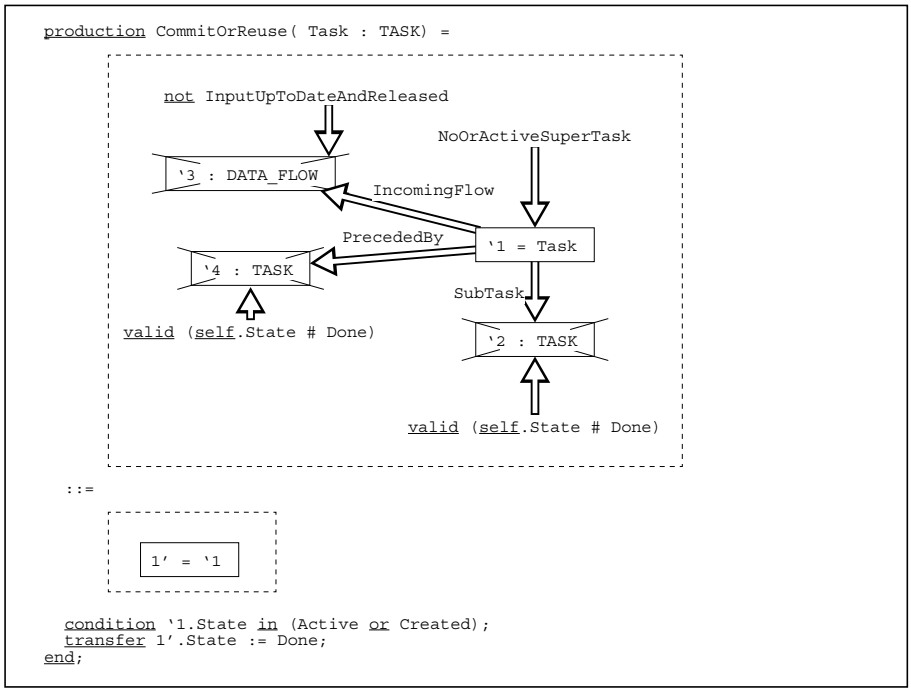


Figure 1.20: `Commit/Reuse` transition

The graph rewrite rule of Figure 1.20 covers both `Commit` and `Reuse` *state transitions*. The condition part checks that these transitions may be performed in the current task state (see also Figure 1.18 on page 34). According to Table 1.2, a complex task may only be moved into the state `Done` if all subtasks are `Done` as well (node `'2`). Furthermore, all predecessor tasks must have terminated successfully (node `'4`; see Table 1.3). In addition, the current task must either be a root task or must have an active supertask (restriction `NoOrActiveSupertask`). Finally, all inputs must be up to date and released, i.e., the task must have seen the final value of each input (node `'3` and the corresponding restriction). Due to space limitations, we omit the definitions of the restrictions and paths used in this graph rewrite rule (and the following ones).

Workspace management is in charge of consuming inputs and producing outputs (see also Figure 1.8 on page 16). These operations may only be performed by `Active` tasks (see Table 1.4). `Consume` (Figure 1.21), which is applied to an incoming data flow (node `'1`) with enabled propagation (condition part),

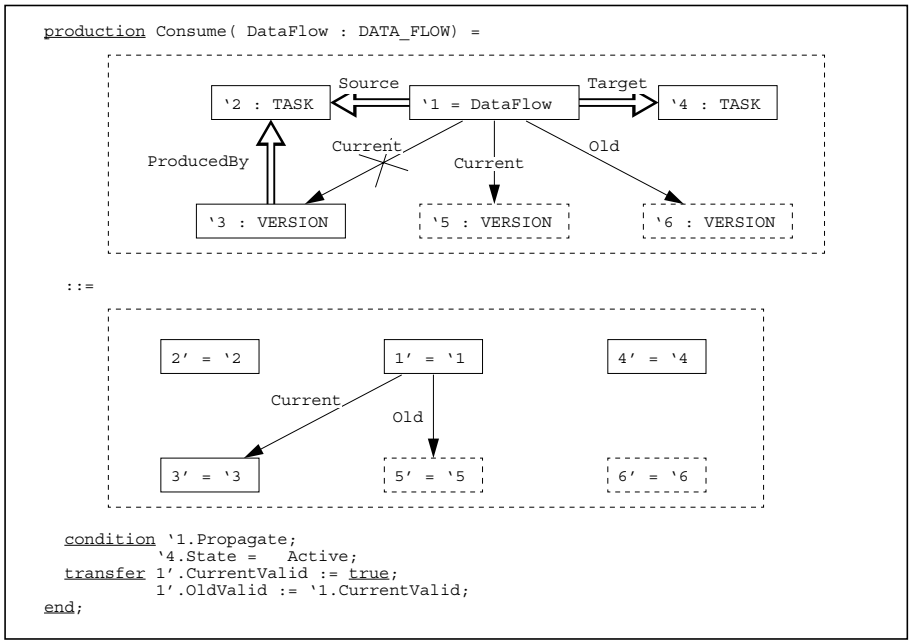
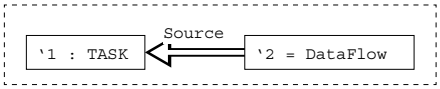


Figure 1.21: Consumption of an input version

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.

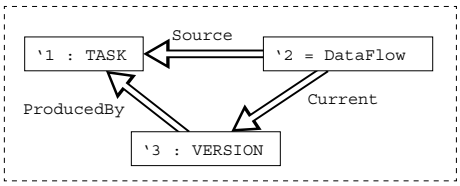
Release management is concerned with the release states of task outputs. For example, `ReleaseToTask` (Figure 1.22) releases an output selectively to a specific successor task. The auxiliary operation `AuxReleaseToTask` checks that the release state is `PartiallyReleased` and that propagation is currently disabled. Normally, the releasing task is `Active`; however, handling of feedback requires manipulation of release states of iterated tasks having been moved back to the state `Created`. After propagation has been enabled, `UpdateCurrentValid` may have to be applied in order to update the value of the `CurrentValid` attribute (which has been used in the `CommitReuse` rule explained above). If the current input version is up to date, `CurrentValid` must have the same value as `Propagate`. Otherwise, no action is necessary. Conditional application is achieved through a `choose` statement which tries to

```
transaction ReleaseToTask( DataFlow : DATA_FLOW) =
    AuxReleaseToTask ( DataFlow )
  & choose
      UpdateCurrentValid ( DataFlow )
    else
      skip
    end
end;

production AuxReleaseToTask( DataFlow : DATA_FLOW) =
```

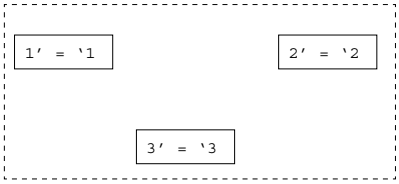



```
  ::=
```


```
  condition `1.State in (Active or Created);
            `1.ReleaseState = PartiallyReleased;
            not `2.Propagate;
  transfer 2'.Propagate := true;
end;

production UpdateCurrentValid( DataFlow : DATA_FLOW) =
```




```
  ::=
```




```
  condition `2.CurrentValid # `2.Propagate;
  transfer 2'.CurrentValid := `2.Propagate;
end;
```

Figure 1.22: Selective release

apply `UpdateCurrentValid`. In case of failure, the `skip` statement is executed, which is always successful.

1.4.3 Model Adaptation

So far, the specification has been independent of a specific application domain. The domain-independent part of the specification is called *generic model*. In the following, we discuss how the generic model is adapted to a specific application domain. The result of such an adaptation is called *specific model*.
In general, adaptation of the generic management model may involve the definition of domain-specific types of objects, relationships, and attributes, as well as domain-specific constraints and operations. Within the SUKITS project, however, we have studied adaptation only to a limited extent, mainly focusing on the definition of a domain-specific schema for product management. Due to the product-centered approach, this schema is used to adapt process management as well. For example, each component type in a configuration type corresponds to a task type. Below, we restrict ourselves to discussing this rather simple kind of adaptation; for a more comprehensive discussion, the reader is referred to [38].

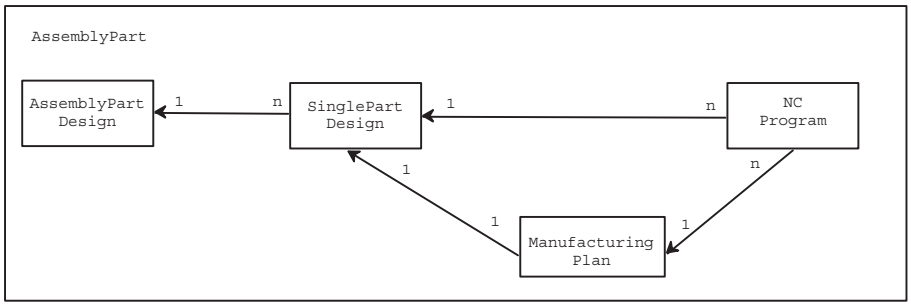


Figure 1.23: Specific schema for product management

Figure 1.23 shows an ER-like diagram which defines a *domain-specific schema* for product management[e]. Boxes and arrows represent entity and relationship types, respectively. An assembly part is described by an `AssemblyPartDesign`. In addition, for each single part occurring in the assembly part there is a corresponding `SinglePartDesign` whose manufacturing process is described in a

[e] The schema is not complete with respect to the examples studied earlier; e.g., it does not include entity types for part lists or raw part designs.

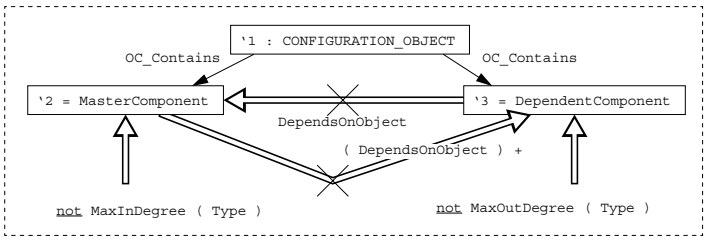
`ManufacturingPlan`. Finally, `NCProgram` stands for the NC programs controlling the manufacturing process of a single part.
We require that the operations provided by the customized product management system be constrained by the domain-specific schema. For example, the product management system should only offer operations for creating objects instantiated from the specific entity types. Similarly, when a dependency is created, it has to be instantiated from a specific relationship type and needs to connect correctly typed objects, respecting the given cardinality constraints.
To achieve this, we could simply write new graph transformations taking the domain-specific schema into account. This means that we introduce one operation for each entity type and one operation for each relationship type occurring in the diagram of Figure 1.23. For example, an operation `CreateNCMPDependency` would create dependencies between NC programs and manufacturing plan.
The disadvantage of this approach is that it results in a large number of specific graph transformations. This can be avoided as follows: To adapt the generic model, the graph schema is extended with domain-specific types and constraints. Graph transformations are written in a generic way, i.e., they are supplied with type parameters and check certain attributes and constraints (*generic operations*). Therefore, the graph transformations need not be modified in order to adapt the specification. For example, we may use a single operation for creating all types of dependencies introduced in Figure 1.23.
The *stratified type system* of PROGRES makes it possible to separate clearly between generic and specific model. Node classes are defined in the generic model (e.g., node class `DOCUMENT_OBJECT`). Node types are then introduced in the specific model (e.g., node type `SinglePartDesign`).
Generic operations are obtained in the following way: Each operation may be applied to instances of multiple types (polymorphism). For nodes to be created, their types are passed as parameters. Furthermore, ER-like constraints are represented by *type-level attributes*[f], i.e., attributes attached to types rather than to instances. Thus, operations are adapted by extending the graph schema. No "code" has to be written for routine adaptations; however, in general it may still be necessary to add further domain-specific operations.
Figure 1.24 shows the definition of type-level attributes for object dependencies. Type-level attributes are introduced at the level of node classes and receive actual values at the level of node types. `MasterTypes` and `DependentTypes` denote the types of master and dependent object components, respectively. `MasterAtMostOnce` and `DependentAtMostOnce` indicate whether a component may have multiple incoming and outgoing dependencies, respectively. In this

[f] Type-level attributes are called meta attributes in PROGRES.

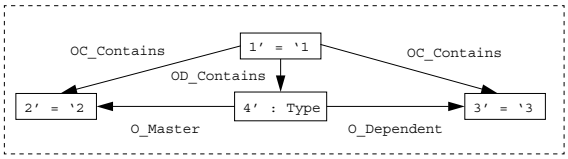
```
node class OBJECT_DEPENDENCY is a NODE
  meta
    MasterTypes : type in OBJECT_COMPONENT [1:n];
    DependentTypes : type in OBJECT_COMPONENT [1:n];
    MasterAtMostOnce : boolean;
    DependentAtMostOnce : boolean;
  ...
end;

node type NC_MP_Dependency : OBJECT_DEPENDENCY
  redef meta
    MasterTypes := ManufacturingPlan ;
    DependentTypes := NCProgram ;
    MasterAtMostOnce := false ;
    DependentAtMostOnce := true ;
end;
```

Figure 1.24: Type-level attributes

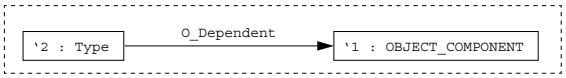
```
production CreateObjectDependency
  ( DependentComponent, MasterComponent : OBJECT_COMPONENT ;
    Type : type in OBJECT_DEPENDENCY ;
    out NewDependency : Type)
 =
```




```
  ::=
```




```
  condition Type in `1.ObjectDependencyTypes;
            `2.type in Type.MasterTypes;
            `3.type in Type.DependentTypes;
  return NewDependency := 4';
end;

restriction MaxOutDegree( Type : type in OBJECT_DEPENDENCY)
  : OBJECT_COMPONENT =
  `1 in
```




```
  condition Type.DependentAtMostOnce;
end;
```

Figure 1.25: Creation of an object dependency

way, upper bounds of cardinalities are represented. The declaration of the node type `NC_MP_Dependency` illustrates how these type-level attributes are defined for dependencies from NC programs to manufacturing plans (see also Figure 1.23).
The corresponding generic graph rewrite rule (Figure 1.25) checks several domain-independent constraints (see Table 1.1 on page 25): Master and dependent component must belong to the same configuration object (Constraint 5), they must not be connected by a dependency (4), and insertion of the dependency must not create a cycle (6). The input parameter `Type` denotes the type of the dependency to be created[g]. The condition part checks that domain-specific constraints for type combinations are not violated. Finally, the negated restrictions `MaxInDegree` and `MaxOutDegree` (nodes `'2` and `'3`, respectively) prevent cardinality overflows.

1.5 Conclusion

We have presented a case study demonstrating that graph rewriting provides a well-suited foundation for specifying the data and operations of an integrated system managing products, processes, and resources in the application area of mechanical engineering. To this end, we have described cut-outs of the PROGRES specification of the management model developed in the SUKITS project. As we have demonstrated, the specification exploits virtually the full range of language constructs for defining graph schemas and graph transformations. In particular, PROGRES offers multiple inheritance on node classes, both instance- and type-level (meta) attributes, both intrinsic and derived attributes and relationships, textual and graphical constraints, graph rewrite rules with very flexible graph patterns, and transactions for programming with graph rewrite rules. The overall specification covers about 60 pages PROGRES code.
Being satisfied with the rich facilities offered for specification-in-the-small, we still miss support for *specification-in-the-large*. In the official PROGRES release, a large specification cannot be decomposed into modules with well-defined interfaces. Current work on a module concept for PROGRES is reported in [39]. Moreover, several chapters of this book (**??** – **??**) are devoted to modularization concepts in specification languages.
Another limitation concerns the way we transformed the specification into an implementation. Since we used IPSEN technology (see [40] and Chapter **??**

[g] In the rules of Figure 1.15 on page 30 and Figure 1.16 on page 32, the type of the new node is not passed as parameter; rather, it is determined by a type-level attribute of some neighbor node.

of this volume), we were forced to convert our general graph structures into abstract syntax graphs, i.e., abstract syntax trees augmented with context-sensitive edges. Unfortunately, abstract syntax graphs are not a natural choice for representing version graphs, configuration graphs, and task graphs. Moreover, there was no automatic support for generating the specification from the implementation.

In our current work, we therefore make use of recently developed tools for generating an implementation from a specification [41]. Furthermore, we are extending our work to a novel application domain, namely chemical engineering. This is done in the context of a collaborative research centre called *IMPROVE* ([42], see also Chapter **??** of this volume).

Acknowledgements

The work reported in this chapter was carried out within the SUKITS project which was funded by the Deutsche Forschungsgemeinschaft (DFG) from 1991 to 1997 under the grant Na 134/5-4.

References

1. Richard H. Thayer. Software engineering project management: A top-down view. In Richard H. Thayer, editor, *Tutorial: Software Engineering Project Management*, pages 15–54. IEEE Computer Society Press, Washington, D.C., 1988.
2. Kenneth G. McIntosh. *Engineering Data Management — A Guide to Successful Implementation*. McGraw-Hill, Maidenhead, England, 1995.
3. S. B. Harris. Business strategy and the role of engineering product data management: A literature review and summary of the emerging research questions. *Proceedings of the Institution of Mechanical Engineers, Part B (Journal of Engineering Manufacture)*, 210(B3):207–220, 1996.
4. Walter Tichy, editor. *Configuration Management*, volume 2 of *Trends in Software*. John Wiley & Sons, New York, 1994.
5. Stefan Jablonski and Christoph Bußler. *Workflow Management — Modeling Concepts and Architecture*. International Thomson Publishing, Bonn, 1996.
6. Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors. *Software Process Modelling and Technology*. Advanced Software Development Series. Research Studies Press (John Wiley & Sons), Chichester, UK, 1994.

7. Harold Kerzner. *Project Management: A Systems Approach to Planning, Scheduling, and Controlling*. John Wiley and Sons, New York, 1998.
8. Marc J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4):364–370, December 1975.
9. Stuart I. Feldman. Make — A program for maintaining computer programs. *Software–Practice and Experience*, 9(4):255–265, April 1979.
10. David Leblang. The CM challenge: Configuration management that works. In Tichy [4], pages 1–38.
11. IBM, Böblingen, Germany. *IBM FlowMark: Modeling Workflow*, 2.1 edition, March 1995.
12. 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, February 1993. IEEE Computer Society Press.
13. Manfred Nagl and Bernhard Westfechtel, editors. *Integration von Entwicklungssystemen in Ingenieuranwendungen*. Springer-Verlag, Heidelberg, 1998.
14. Bernhard Westfechtel. Using programmed graph rewriting for the formal specification of a configuration management system. In E. Mayr, G. Schmidt, and G. Tinhofer, editors, *Proceedings WG '94 Workshop on Graph-Theoretic Concepts in Computer Science*, LNCS 903, pages 164–179, Herrsching, Germany, June 1994. Springer-Verlag.
15. Bernhard Westfechtel. A graph-based system for managing configurations of engineering design documents. *International Journal of Software Engineering and Knowledge Engineering*, 6(4):549–583, December 1996.
16. Bernhard Westfechtel. Graph-based models and tools for managing development processes. Habilitation thesis, RWTH Aachen, Germany, 1999.
17. Bernhard Westfechtel. Integrated product and process management for engineering design applications. *Integrated Computer-Aided Engineering*, 3(1):20–35, January 1996.
18. Bernhard Westfechtel. Engineering data and process integration in the SUKITS environment. In Julian Winsor, editor, *Proceedings International Conference on Computer Integrated Manufacturing (ICCIM '95)*, pages 117–124, Singapore, July 1995. World Scientific.
19. Andy Schürr. Introduction to PROGRES, an attribute graph grammar based specification language. In M. Nagl, editor, *Proceedings WG '89 Workshop on Graph-Theoretic Concepts in Computer Science*, LNCS 411, pages 151–165, Castle Rolduc, The Netherlands, June 1989. Springer-Verlag.
20. Andy Schürr. PROGRES: A VHL-language based on graph grammars.

In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, LNCS 532, pages 641–659, Bremen, Germany, March 1990. Springer-Verlag.

21. Andy Schürr. Rapid programming with graph rewrite rules. In *USENIX Symposium on Very High Level Languages*, pages 83–100, Santa Fe, New Mexico, October 1994. USENIX Association.
22. Andy Schürr, Andreas Winter, and Albert Zündorf. Graph grammar engineering with PROGRES. In Wilhelm Schäfer and Pere Botella, editors, *Proceedings of the European Software Engineering Conference (ESEC '95)*, LNCS 989, pages 219–234, Barcelona, Spain, September 1995. Springer-Verlag.
23. Andreas Zamperoni. GRIDS — graph-based integrated development of software: Integrating different perspectives of software engineering. In *Proceedings of the 18th International Conference on Software Engineering*, pages 48–59, Berlin, March 1996. IEEE Computer Society Press.
24. Georg Heidenreich. Ein generisches Organisationsschema der Konfigurationsverwaltung im Software-Engineering. Technical Report 29/13, University of Erlangen, Erlangen, Germany, September 1996.
25. Nico Habermann and David Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, 12(12):1117–1127, December 1986.
26. Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator*. Springer-Verlag, New York, 1988.
27. Klaus Dittrich, Willi Gotthard, and Peter Lockemann. DAMOKLES, a database system for software engineering environments. In Reidar Conradi, Tor M. Didriksen, and Dag H. Wanvik, editors, *Proceedings of the International Workshop on Advanced Programming Environments*, LNCS 244, pages 353–371, Trondheim, June 1986. Springer-Verlag.
28. Theo Härder, Klaus Meyer-Wegener, Bernhard Mitschang, and A. Sikeler. PRIMA — a DBMS prototype supporting engineering applications. In *Proceedings of the 13th Conference on Very Large Databases*, pages 433–442, Brighton, England, September 1987.
29. Lois Wakeman and Jonathan Jowett. *PCTE — The Standard for Open Repositories*. Prentice Hall, Englewood Cliffs, New Jersey, 1993.
30. Peter P. Chen. The entity-relationship model — toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.
31. Francois Bancilhon, Claude Delobel, and Paris Kannelakis, editors. *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann, San Mateo, California, 1992.
32. Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The

Objectstore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

33. Wolfgang Deiters and Volker Gruhn. The FUNSOFT net approach to software process management. *International Journal of Software Engineering and Knowledge Engineering*, 4(2):229–256, 1994.
34. Sergio 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.
35. Hans-Jörg Bullinger and Joachim Warschat, editors. *Concurrent Simultaneous Engineering Systems*. Springer-Verlag, Berlin, Germany, 1996.
36. Andy Schürr. Introduction to the specification language PROGRES. In Nagl [40], pages 248–279.
37. Norbert Kiesel, Andy Schürr, and Bernhard Westfechtel. GRAS, a graph-oriented software engineering database system. *Information Systems*, 20(1):21–51, January 1995.
38. Carl-Arndt Krapp. *An Adaptable Environment for the Management of Development Processes*. PhD thesis, RWTH Aachen, Aachen, Germany, 1998.
39. Andreas Winter and Andy Schürr. Modules and updatable graph views for programmed graph rewriting systems. Technical Report AIB 97-03, RWTH Aachen, Aachen, Germany, 1997.
40. Manfred Nagl, editor. *Building Tightly-Integrated Software Development Environments: The IPSEN Approach*. LNCS 1170. Springer-Verlag, Berlin, Germany, 1996.
41. Peter Heimann, Carl-Arndt Krapp, and Bernhard Westfechtel. An environment for managing software development processes. In *Proceedings of the 8th Conference on Software Engineering Environments*, pages 101–109, Cottbus, Germany, April 1997. IEEE Computer Society Press.
42. Manfred Nagl and Wolfgang Marquardt. SFB-476 IMPROVE: Informatische Unterstützung übergreifender Entwicklungsprozesse in der Verfahrenstechnik. In Matthias Jarke, Klaus Pasedach, and Klaus Pohl, editors, *Informatik '97: Informatik als Innovationsmotor*, Informatik aktuell, pages 143–154, Aachen, Germany, September 1997. Springer-Verlag.