20-35, John Wiley & Sons (January 1996)

Integrated Product and Process Management for Engineering Design Applications

BERNHARD WESTFECHTEL
Lehrstuhl für Informatik III, RWTH Aachen
Ahornstr. 55, D–52074 Aachen

ABSTRACT

SUKITS is a joint project of computer scientists and mechanical engineers which is devoted to the design and implementation of an integrated infrastructure for product development. This infrastructure – the CIM Manager – provides a framework for a posteriori integration of heterogeneous application systems. This paper describes management of engineering products and processes. Product management deals with configurations of interdependent, versioned documents such as designs, manufacturing plans, or NC programs. Management of engineering processes is tightly integrated with product management. A configuration of interdependent documents is interpreted as a net of communicating processes. Concurrent engineering is supported by exchanging pre–releases of intermediate results.

INTRODUCTION

Due to enormous pressures from national and international marketplaces, Computer Integrated Manufacturing has become a tremendously important area of both research and development. In contrast to traditional taylorism which emphasizes division of labour, the purpose of CIM is to integrate all processes carried out within an enterprise (or even across multiple enterprises). However, although progress has definitely been made in this direction, the state of the art is still characterized by islands of automation: while CIM application systems themselves are quite powerful, much of their power is lost due to poor interfaces between them.

In response to these deficiencies, a project called SUKITS (Software and Communication Structures in Technical Systems) was launched by the Technical University of Aachen in fall 1991. SUKITS is a joint effort of computer scientists and mechanical engineers which is devoted to bridging the gaps between islands of automation. This goal is achieved by providing an integrating infrastructure which is called *CIM Manager* (Eversheim et al., 1992). The CIM Manager acts as a framework for a posteriori integration of heterogeneous application systems which have been developed by different vendors, run under different operating systems, and rely on different data management systems (Figure 1).

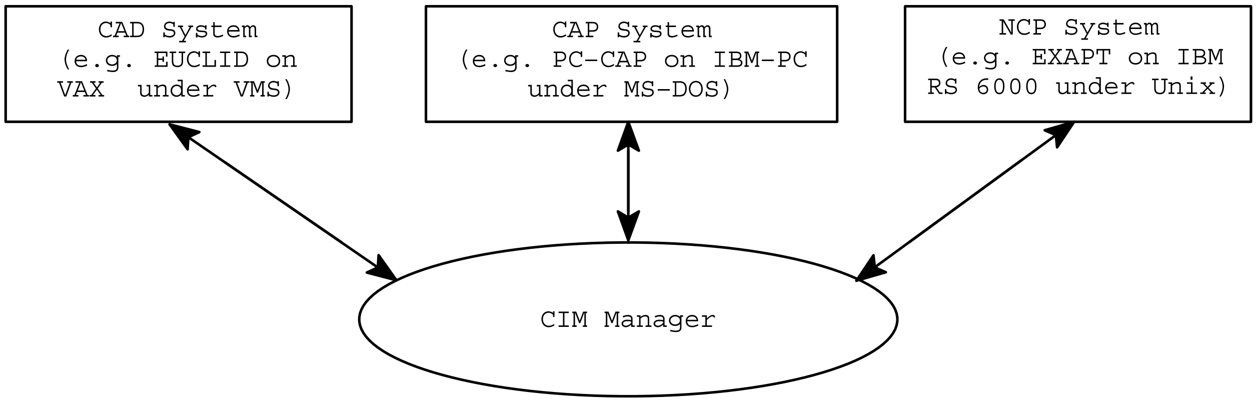
CAD System (e.g. EUCLID on VAX under VMS)
CAP System (e.g. PC-CAP on IBM-PC under MS-DOS)
NCP System (e.g. EXAPT on IBM RS 6000 under Unix)
CIM Manager

Figure 1 A posteriori integration of heterogeneous CIM components

In contrast to more comprehensive projects like e.g. CIM–OSA (Kosanke, 1991), we do not intend to cover the whole area of Computer Integrated Manufacturing. Rather, we focus on *product development*, i.e. on all engineering activities which are concerned with developing descriptions of products and manufacturing processes. The results of such activities are captured in documents such as CAD designs, manufacturing plans, or NC programs. Documents may contain both product and (manufacturing) process data. So far, we have not investigated those phases of the product life cycle which follow product development, namely production planning and production. On the other hand, our work exceeds the CIM area because we reuse our integrating infrastructure in other domains, specifically in software engineering (Westfechtel, 1995).

In this paper, we focus on two aspects of the SUKITS project, namely product and process management and their integration. *Product management* refers to managing documents which evolve into multiple versions, are connected by manifold dependencies, and are combined in many configurations. Thus, the term “product” refers to descriptions of manufactured products and manufacturing processes rather than to manufactured products themselves. Our approach to product management is characterized by the following features:

- The product management model is domain–independent, i.e. it may be applied to different domains of engineering design (e.g. mechanical and software engineering).
- Since product management has been designed for a broad range of applications, its underlying model is based on a sparse set of reusable concepts, namely composition hierarchies, versioning, and dependencies.
- Rather than supporting these concepts in isolation, product management integrates them into a coherent framework. In particular, we provide for uniform versioning of both documents and document groups (revisions and configurations, respectively), nested configurations, and control of consistency between interdependent, versioned components of configurations.
- Product management is adaptable to a certain application domain. Product management operations are tailored by an ER–like schema which defines object and relation types specific to a certain application domain. In this way, the functionality of the domain–independent kernel is enriched so that domain–specific knowledge is taken into account, as well.

Process management refers to managing engineering activities which are carried out in order to deliver documents describing products and manufacturing processes. In this paper, we are concerned with management of engineering processes rather than control of manufacturing processes. The goal of process management is to assist engineers in keeping documents describing products and manufacturing processes consistent with each other. To this end, process management supports close cooperation between engineers operating in different work areas of product development:

- Like conventional approaches which rely e.g. on net plans or Petri nets, processes are arranged in a process net which represents data and control flow dependencies.
- Since engineering processes are highly creative activities, it is in many cases impossible to predict all processes to be carried out from the very beginning. Therefore, process management supports incremental extensions of process nets.
- Furthermore, process management takes the iterative nature of engineering processes into account. Only rarely can engineering processes be executed once and for all, delivering a perfect result at once. Rather, errors will be detected in subsequent processes so that iterations need to be performed to improve on the initial result.
- In order to shorten development time drastically, approaches such as *concurrent engineering* (Reddy, Srinivas, and Jagannathan, 1993) are currently a hot topic. Process management supports concurrent engineering inasmuch as intermediate results may be pre–released to dependent processes as soon as possible. Thus, the start of a process need not be delayed until all preceding processes have been executed completely.
- There is a very close integration between product and process management. By interpreting a configuration of interdependent components as a process net, we avoid the conceptual overhead which is implied by separating product and process structure and keeping the latter consistent with the former.

The rest of this paper is organized as follows: Firstly, we describe our approach to product management. The next section is devoted to process management. Subsequently, we present the SUKITS prototype for product and process management. Finally, we discuss related approaches and conclude the paper by giving an outlook on future work.

PRODUCT MANAGEMENT

Product management deals with engineering design documents (briefly denoted as *documents* in the sequel). Documents are artifacts generated or consumed during the engineering 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.

Logically related documents are collected in *document groups*. For example, a document group may comprise all documents describing a single part and its manufacturing process. Although all documents of a group might be created by the same engineer, a document group usually serves as a work context for integrating documents produced by multiple engineers. In general, document groups may be nested so that they form a composition hierarchy of arbitrary depth.

Components of a document group are related by various kinds of *dependencies*. Such dependencies may either connect components belonging to the same work area (e.g. dependencies between descriptions of components of an assembly part), or they may cross work area boundaries (e.g. dependencies between designs and manufacturing plans). Accurate management of dependencies is an essential prerequisite for consistency control, i.e. for keeping interdependent components consistent with each other. To this end, tools are needed which operate on the fine–grained level, i.e. on the contents of documents. Since product management follows a coarse–grained approach, the CIM Manager itself cannot perform consistency control on the fine–grained level. However, it provides a reusable framework for embedding domain–specific tools operating on the fine–grained level.

During its evolution history, both documents and document groups evolve into multiple *versions*. Versions may be regarded as snapshots recorded at appropriate points in time. The reasons for managing multiple versions of an object (instead of just its current state) are manifold: reuse of versions, maintenance of old versions having already been delivered to customers, storage of back–up versions, or support of change management (e.g. through a `diff` analysis figuring out what has changed with respect to an old version).

The CIM Manager supports uniform versioning of both documents and document groups. Therefore, we subsume both notions under the generic term *object*. Each object has an evolution history which is represented by a graph of versions interconnected by history relations. Versions of documents and document groups are denoted as *revisions* and *configurations*, respectively. Due to uniform versioning, all benefits gained from version control (e.g. reuse, change management) may be exploited on arbitrary levels of the composition hierarchy.

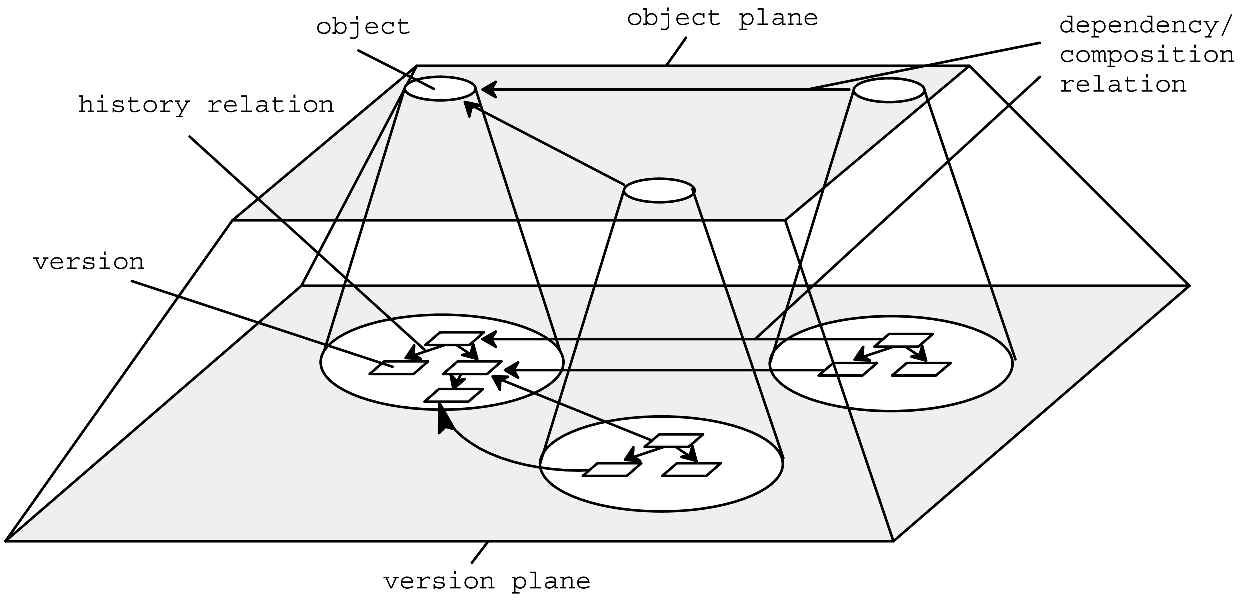

Figure 2 Object and version plane

According to the distinction between objects and their versions, the product model distinguishes between an *object plane* and a *version plane* (Figure 2). An object represents a set of versions. The version plane refines the object plane: each object is refined into its versions, and each relation between two objects is refined into relations between corresponding versions. Furthermore, history relations 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 revisions and configurations as well as their mutual relations.

Objects, versions, and their interrelations are formally represented by *graphs*. In order to structure the information contained in such graphs, the product model distinguishes between different kinds of interrelated subgraphs, namely version, configuration, and document group graphs, respectively. Each subgraph provides a view on the underlying database displaying the evolution history of an object, the structure of a configuration, and the structure of a document group, respectively.

A *version graph* (Figure 3) consists of versions which are connected by history relations. A history relation 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. 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 model allows for all structures shown in Figure 3; it merely excludes cycles in history relations. Variants may either be represented by branches or by separated subgraphs.

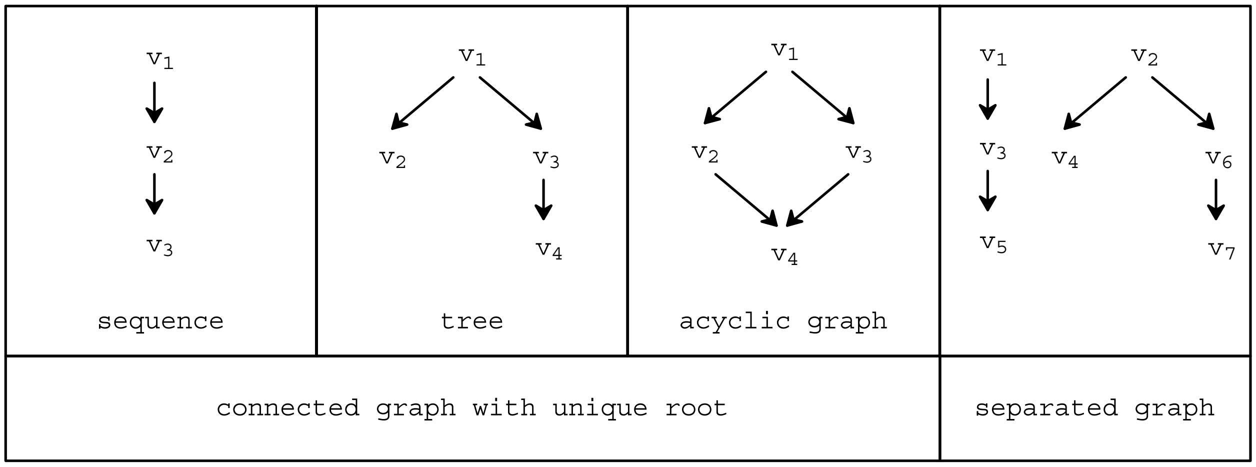

Figure 3 Structures of version graphs

A *configuration graph* contains version components and their mutual dependencies. As we have already mentioned above, configurations are versions, too. Therefore, configurations of document groups may be employed as units of reuse in the same way as revisions of documents. As a simple example, the top half of Figure 4 shows three configurations of a document group gathering some documents for a certain single part (e.g. a shaft). c_1 contains initial revisions of a CAD design, a manufacturing plan, and two NC programs. The manufacturing plan depends on the design, and the NC programs depend on both the manufacturing plan and the design. In c_2, the manufacturing plan has been changed, resulting in corresponding modifications to both NC programs. The transition from c_2 to c_3 involves a structural change introducing another NC program so that the single part is manufactured in three consecutive steps.

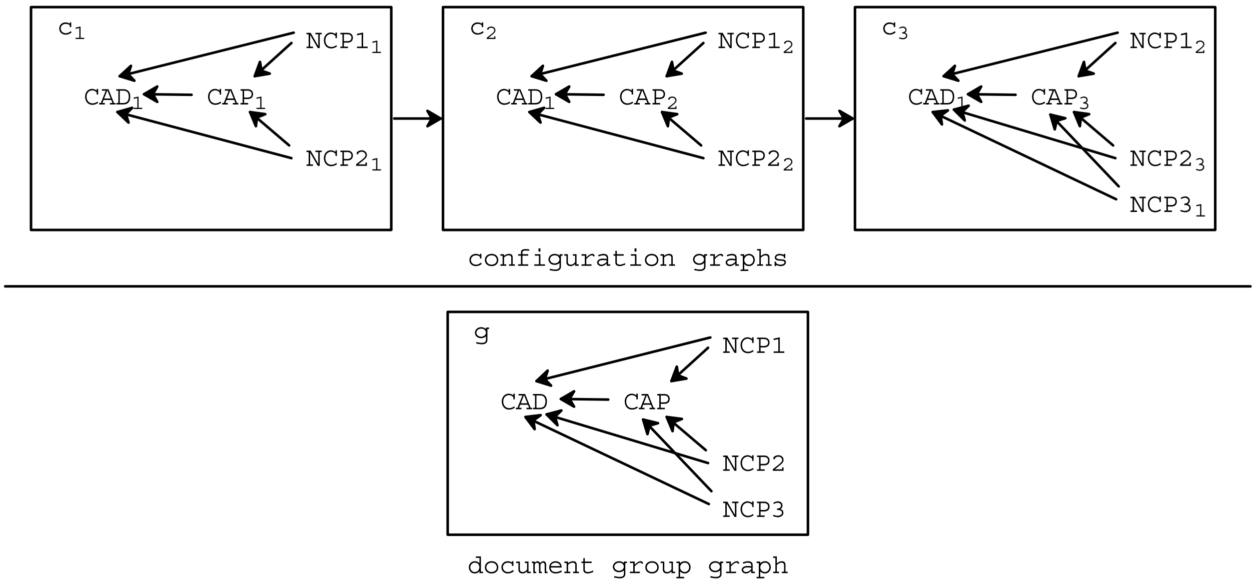

Figure 4 Document group graphs and configuration graphs

A configuration graph represents a snapshot of the evolution of a document group. All configurations of a document group constitute a family of closely related versions of product descriptions. Structural properties of the whole family rather than individual members are expressed by a *document group graph* (bottom half of Figure 4). The document group graph may be viewed as a union of all configuration graphs: for each version component (dependency) contained in a configuration graph, a corresponding object component (dependency) must exist in the document group graph. The document group graph contains varying parts, i.e. components and dependencies which do not belong to the intersection of all configuration graphs. Large varying parts result in a complex and imprecise document group graph and indicate an ill–structured, heterogeneous family. On the other hand, we do not enforce identically structured configurations because this is too restrictive in practice.

The CIM Manager provides comfortable commands for maintaining the relations between document group graph and configuration graphs. Top–down commands are used to generate (parts of) configuration graphs from the document group graph. Conversely, bottom–up commands are used to complete the document group graph after having extended some configuration graph. Furthermore, various analysis commands are provided which e.g. show varying parts of a document group or compare configurations of the same document group.

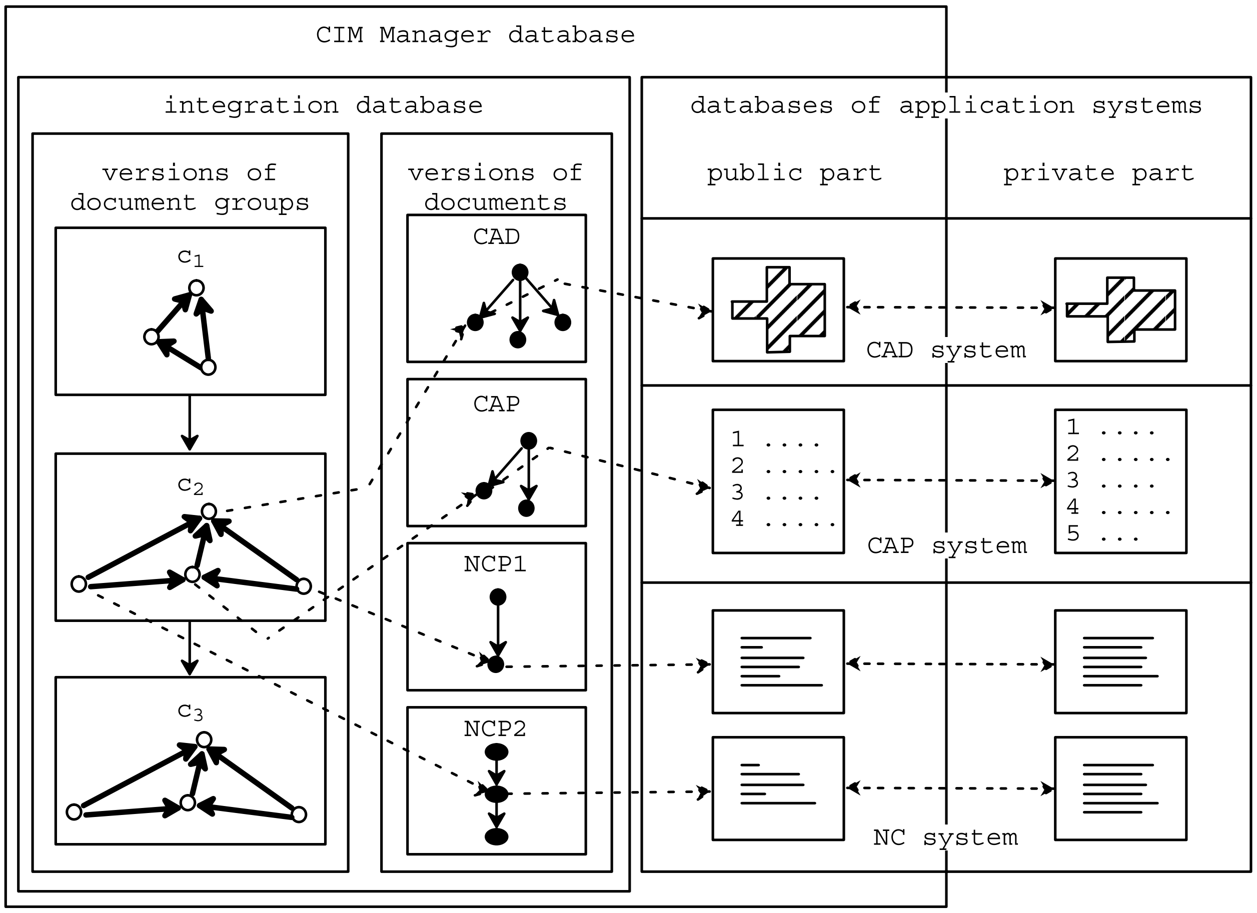

Figure 5 Data integration through the CIM Manager

The SUKITS approach to data integration (Schwartz and Westfechtel, 1993) is summarized in Figure 5. According to the constraints imposed by a posteriori integration of heterogeneous application systems, we follow a *meta data approach*. The integration database of the CIM Manager manages configuration hierarchies whose leaves correspond to revisions of documents. Instead of storing their contents centrally, the integration database maintains references into the databases of application systems. Whenever possible, an application database is separated into a public part, which is controlled by the CIM Manager, and a private part, which remains under control of the application system. Data are exchanged between public and private part by `Checkout` and `Checkin` operations. In particular, this approach may be applied to application systems which support export and import of files, possibly via a neutral data format such as IGES or STEP. However, it does not work in case of completely closed application systems where the CIM Manager has no means to access the data maintained by the application. For a closed system, the CIM Manager can merely maintain some sort of reference without providing for access control and referential integrity.

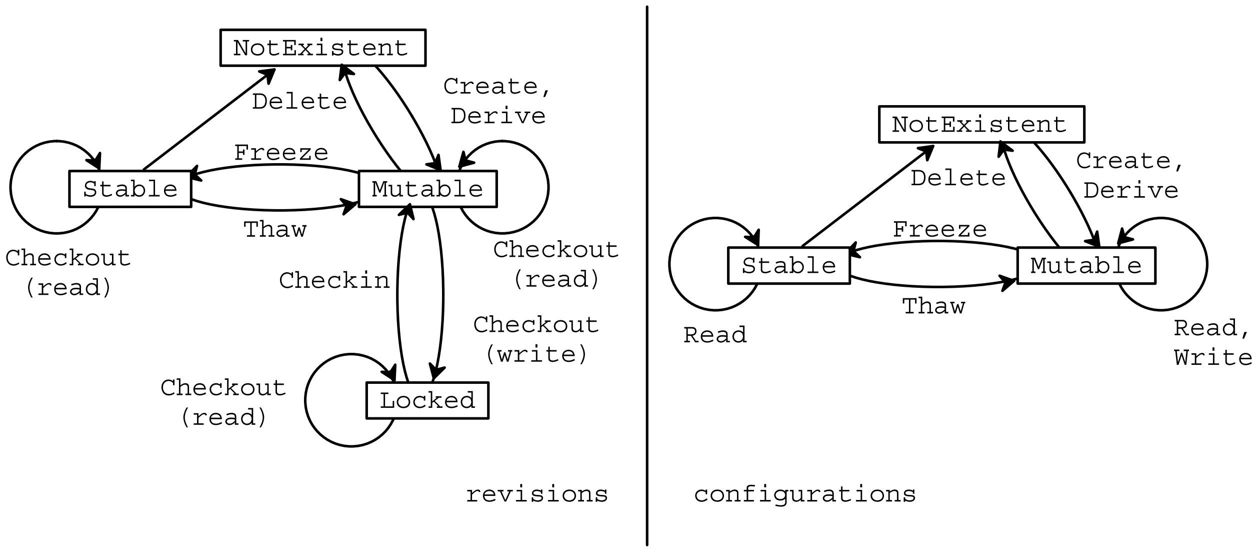

Figure 6 State transition diagrams for revisions and configurations

Figure 6 shows *state transition diagrams* for revisions and configurations. Each transition corresponds to an application of a certain operation. In order to provide for uniform versioning of configurations and revisions, the state transition diagrams have been unified as far as possible. In particular, a `Stable` attribute, which indicates whether the version may be modified, is attached to both revisions and configurations; `Freeze` and `Thaw` assign values `true` and `false` to the `Stable` attribute, respectively. However, there are still some differences between the state transition diagrams: Since revisions are manipulated outside the CIM Manager database, `Checkout` and `Checkin` are needed for data exchange. On the other hand, configurations are manipulated in place, i.e. within the integration database. Therefore, `Checkout` and `Checkin` are not provided on configuration level.

Figure 7 describes the *schema* for product management by an ER–like diagram. A complex entity type is represented by a labeled rectangle which surrounds a subdiagram describing its contents. A double–framed box denotes a component type occurring with cardinality many. Binary, directed relations are drawn as diamonds which are connected to source and target by solid lines. Finally, diamonds with adjacent dashed lines represent inheritance relations.

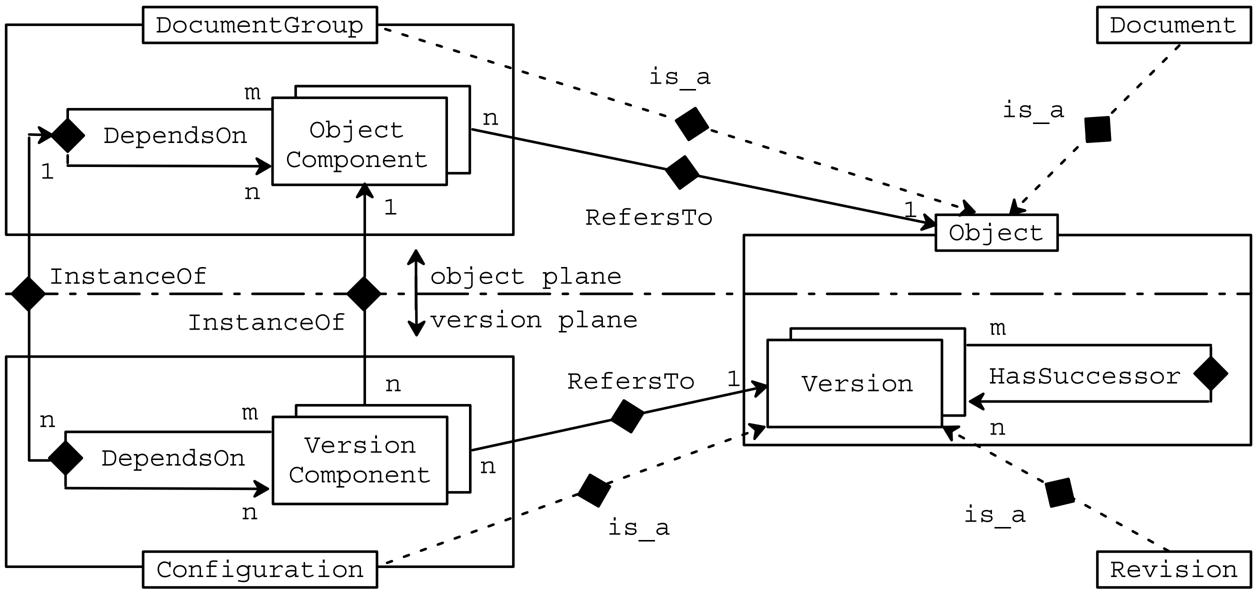

Figure 7 Schema diagram

The diagram reads as follows: Each Object consists of a set of Versions connected by HasSuccessor relations. DocumentGroup/Document and Configuration/Revision are subtypes of Object and Version, respectively. Each DocumentGroup (Configuration) contains ObjectComponents (VersionComponents) which are connected by DependsOn relations. InstanceOf relations connect elements of Configurations and DocumentGroups. Finally, ObjectComponent and VersionComponent represent applied occurrences of Object and Version, respectively (RefersTo relations).

Entities and relations carry attributes (which are not shown in the diagram). Typical examples are: object name, version number, creation date, last modification date, etc. In the following, we focus on attributes which model different aspects of *consistency control* (all of these attributes have the type boolean):

- *Internal consistency* refers to the local consistency of a version. Internal consistency does not take relations to other versions into account and is therefore modeled as an attribute attached to Version entities. For example, a revision of an NC program is denoted as 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. Therefore, external consistency is modeled as an attribute attached to DependsOn relations. For example, a revision of an NC program is consistent with a revision of a CAD design if it conforms to the geometric data specified in the design.
- Finally, *configuration consistency* refers to the consistency of a version component with respect to a certain configuration. Therefore, configuration consistency is modeled as an attribute attached to VersionComponent entities. Configuration consistency implies that the corre-

sponding version is internally consistent, and that it is externally consistent with all master components. For example, a revision of an NC program is consistent with respect to a certain configuration if it is both internally consistent and externally consistent with the design contained in this configuration.

The schema presented in Figure 7 defines the *meta model* for product management. The meta model is a domain–independent model which is built–in and therefore cannot be changed by the user. However, the meta model may be adapted to a certain application domain by specifying a *concrete model*. Types defined in such a concrete model are instances of meta types defined in the meta model. At run–time, entities and relations are instantiated from concrete types. By specifying a concrete model, the CIM Manager is adapted so that it takes the constraints of a certain application domain into account. Without a concrete model, the CIM Manager would know nothing about its application domain, and the user would have to manage application–specific constraints by himself.

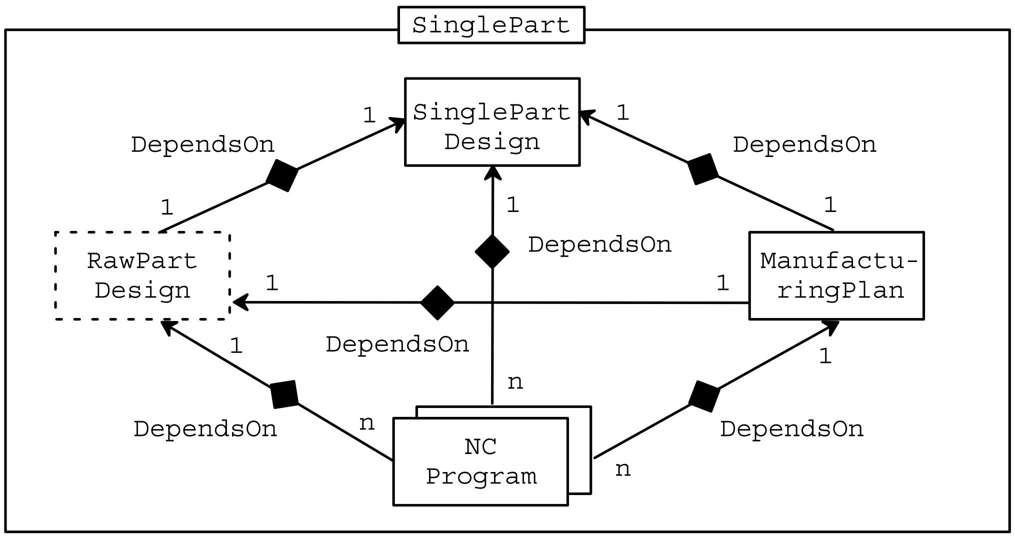

Figure 8 Example for the specification of a concrete model

An example of a concrete model is given in Figure 8. In addition to the symbols occurring in Figure 7, single–framed boxes indicate single components, and dashed boxes represent optional components. The figure describes a document group type `SinglePart` consisting of exactly one `SinglePartDesign`, exactly one `ManufacturingPlan`, an optional `RawPartDesign`, and at least one `NCProgram`. Components are related by dependencies which should be obvious from the examples given so far. While the meta model for document groups is very general, but also rather poor, the concrete model is more specific and enriches the meta model by defining application–specific entity and relation types.

PROCESS MANAGEMENT

Control of manufacturing processes has been studied for a long time. Manufacturing processes may be described rather precisely, and formal methods such as Petri nets have been applied successfully in this area for a long time. On the other hand, engineering processes are highly creative and therefore are more difficult to handle formally. Nevertheless, effective work flow management plays a key role in accelerating product development so that development cycles may be shortened significantly.

In the SUKITS project, we have developed an approach to process management which is closely integrated with product management. The term *product–centered process management* emphasizes this point. By interpreting a configuration of interdependent components as a process net, we avoid the conceptual overhead which is implied by separating product and process structure and keeping the latter consistent with the former. Basically, each component of a configuration corresponds to a process which has to produce this component (i.e. a *component process*). Dependencies between components are mapped onto *data flows* between component processes. Data flows and dependencies have opposite directions: While a dependency emanates from a dependent component and ends at a master component, the corresponding data flow starts at the master process and ends at the dependent process.

Process nets are not represented separately from configurations. Rather, we enrich configurations by all information which is required for process management. In particular, this implies attachment of process–related attributes to components of configurations (e.g. execution state of a component process). Data flows need not be represented explicitly because they may be defined as derived relations. In the current section, we are more interested in data flows than in dependencies. Therefore, data flows are shown in diagrams rather than dependencies. Processes are identified by the name of their result.

Process management must cope with *dynamic evolution* of process nets. Only rarely may a process net be built up completely before execution is started. In many cases, decisions are made during execution which determine how to proceed. Either these decisions cannot be anticipated in their totality, or taking all possible execution paths into account yields a huge and complex process net which is hard to understand and to follow. Therefore, it is essential to support modifications of process nets during execution.

A priori knowledge about process nets is specified in a concrete model such as the one given in Figure 8. By means of cardinalities, we may distinguish between optional and mandatory components, components occurring at most once, and components which may occur more than once. Similarly, we may classify dependencies between components. As a consequence, we cannot predict uniquely which processes will have to be carried out, how they will be connected by data flows, and how many instances of a certain process type will be created dynamically. In general, construction and modification of a process net requires manual steps to be carried out by the project manager and/or technical staff. The concrete model provides some guidance and control; furthermore, net modifications can be automated partially by means of tools operating on the fine–grained level (e.g. creation of NC programming processes based on an analysis of the manufacturing plan).

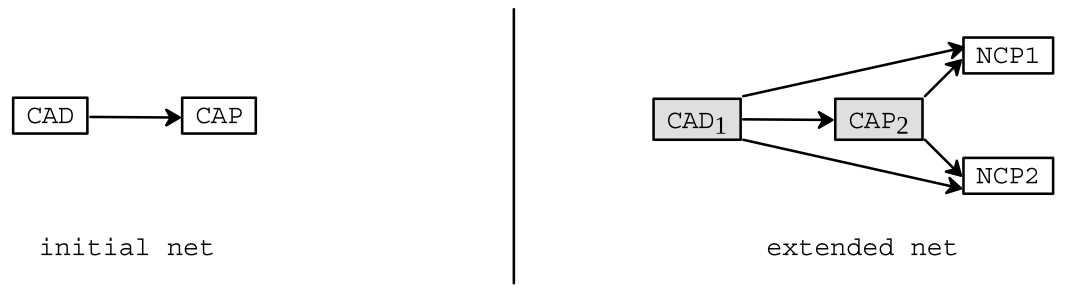

Figure 9 Extension of process nets

Process nets are *extended incrementally* as execution proceeds. A simple example is presented in Figure 9. The left–hand side of the figure shows an initial net which contains a design and a planning process which are connected by a data flow. The processes are identified by the names of the components which have to be developed (CAD and CAP, respectively). Note that the names are not qualified by indices. This means that the corresponding components are not yet bound to a specific version. The product management model supports partially bound configurations in which some components are not yet bound to a specific version. The right–hand side of Figure 9 shows the state of the net after design and planning have been completed. The net has been extended with two NC programming processes which have been defined in the manufacturing plan. Terminated processes are displayed in grey; their indices correspond to the numbers of the versions which they have produced.

The “classical”, conservative rule for defining the execution order states that a process can be started only after all master processes 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 processes may be executed concurrently even if they are connected by data flows. Our approach to process management supports *concurrent engineering* inasmuch as intermediate results may be pre–released to dependent processes as soon as possible.

To this end, each process maintains a *release set* indicating which dependent processes may consume the result produced so far. An “all or nothing” approach to releasing results is too coarse–grained because it depends on the kind of a dependent process when data should be propagated from master to dependent. Therefore, the release set of a process contains the types of component processes to which its result has been (pre–) released.

Figure 10 illustrates the release strategy by means of a simple example. Boxes representing processes are subdivided into two parts. The process name is given in the upper part; the lower part contains the process state and the release set (note that names of component types are used to denote process types). After a draft design has been prepared, this intermediate result has been passed to the planning process so that design and planning process may proceed concurrently. The planning process has in turn released the preliminary manufacturing plan to NC programming. However, NC programming cannot be started yet because geometric data produced in design are still too imprecise (i.e. no data are propagated along dashed flows).

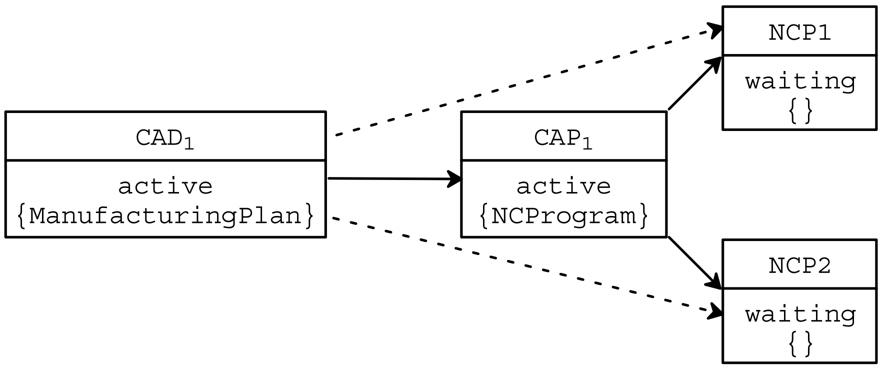

Figure 10 Concurrent execution of master and dependent process

Pre–releases may even be controlled in a more fine–grained way by boolean *propagation attributes* which are attached to data flows. A dependent process may consume data produced by a master process only if its type is included in the master's release set and propagation is enabled along the flow (value `true` of the propagation attribute). For example, consider a draft design of an assembly part which contains single parts `A` and `B` (and potentially further components). The draft design has already been pre–released to the design process for `A`. On the other hand, the design process for `B` cannot start yet because the assembly part design is still too incomplete and immature with respect to `B`.

In order to propagate pre–releases as early as desirable, we relax the *activation condition,* i.e. the predicate which has to be fulfilled when a process is started. Unlike the conservative approach, master processes need not be terminated in order to start a dependent process. To control activation, a boolean attribute `NeededForActivation` is attached to data flows. If the attribute carries the value `true`, the master process must have pre–released its result to the dependent process. `NeededForActivation` allows for tailoring process activations in case of multiple inputs. In general, waiting for pre–releases of all inputs may severely impede concurrency.

Concurrent engineering does not come for free: we have to pay the price of increased complexity which is caused by *changing inputs*. The conservative approach provides for a stable work context of each process. Since all master processes have already been terminated, each active process accesses inputs which will not be modified during process execution. In our model, inputs may change if a dependent and a master process are executed concurrently. An active process is notified of all changes to its inputs. In general, the following changes to inputs have to be taken into account:

- The master process modifies its result, e.g. by creating a successor version or by overriding the current version.
- The master process releases its result.
- The master process revokes its release. Since the activation condition of the dependent process may be violated now, the dependent process may have to be blocked. A blocked process may continue execution only when its activation condition is fulfilled again.

Revoking a release may have to be performed in case of feedbacks. For example, in Figure 10 the author of the manufacturing plan might detect that the single part has not been designed for production, i.e. constraints of the manufacturing process have not been taken into account. Therefore, pre–release of the design is revoked, and planning is blocked until a new release is available.

So far, we have not considered structural changes, i.e. creation or deletion of incoming data flows. Such changes affect the definition of a process, i.e. the specification of its input/output behaviour. Since structural changes have a more fundamental impact than non–structural ones, we require to interrupt the affected process. In this way, we avoid the chaos which may result from concurrent definition and execution of the same process. As soon as structural changes have been finished, process execution may be resumed provided that the activation condition is fulfilled. Otherwise, the process remains blocked until its required inputs are available.

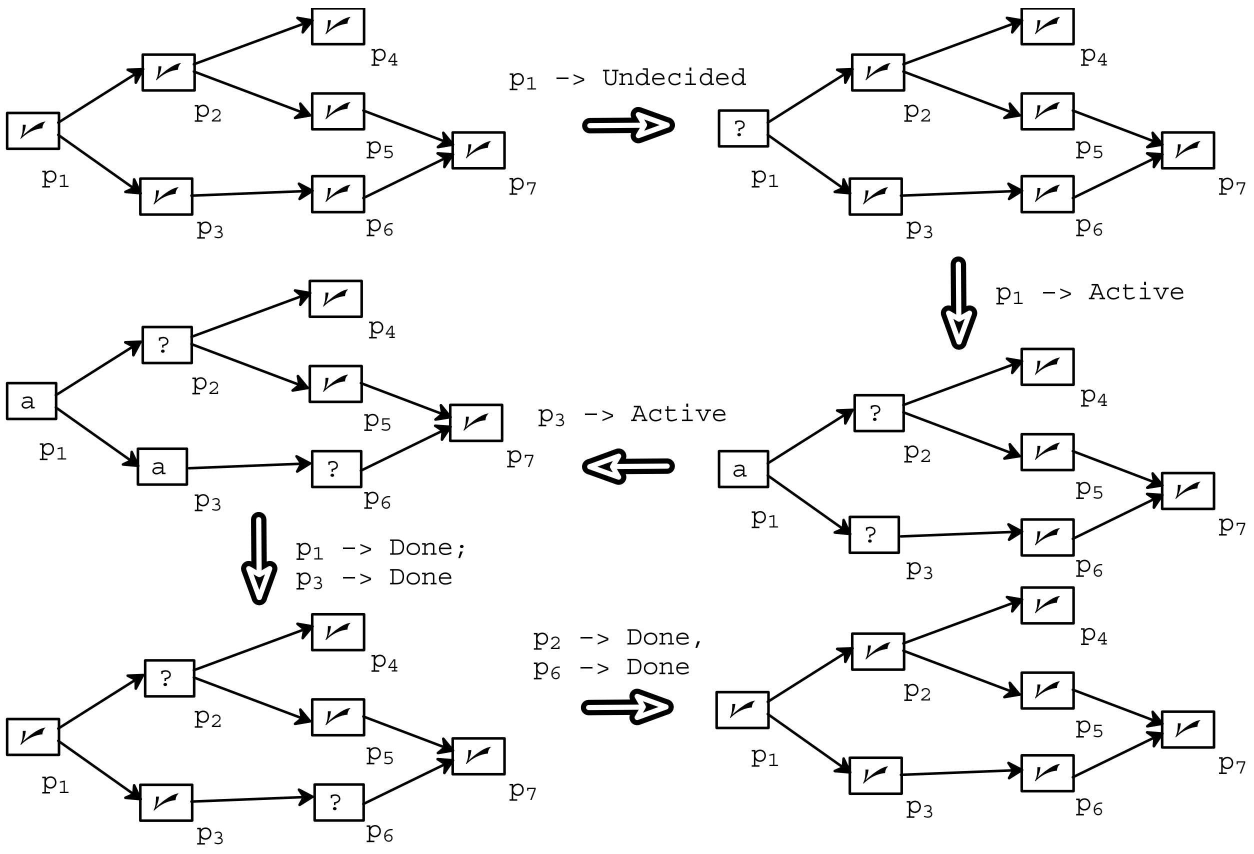

Figure 11 Step–wise change propagation

So far, we have considered activation of processes and interactions between concurrently active processes. However, we have not yet discussed termination of processes. In order to increase the probability that the process need not be reactivated later on, we enforce a *conservative termination condition*:

- Each master process must have been terminated, as well.
- The release set is checked for completeness.
- Propagation must be enabled for all outgoing data flows.
- The version produced by the current process must be frozen.
- Furthermore, the version must be configuration consistent (see previous section).

Unfortunately, we cannot exclude feedbacks from dependent processes after termination. This means that we have to allow for *reactivation* of terminated processes. If the state `Done` were treated as a sink state, feedbacks could not be handled if anyone commits his work too early. For example, after having terminated design and planning in Figure 10, a design error might be detected during NC programming. In order to handle this error, the design process needs to be activated once again.

Reactivation of terminated processes raises a couple of problems. First of all, we have to determine the follow–up state of a process which leaves state `Done`. One possible choice is the initial state of the state transition diagram for processes (see Figure 12 which will be explained comprehensively later on). This approach suffers from the drawback that the process will eventually be forced through the state `Active`, i.e. it will eventually be activated whenever it leaves state `Done`. We have devised a more cautious solution which introduces an intermediate state `Undecided`. This allows for postponing the decision how to proceed. For example, in Figure 10 the NC programmer may be convinced that the design contains an error, and the designer may be proud of the design which he has produced. In such a situation, the design process is moved into the state `Undecided`, and the final decision how to proceed is made later on.

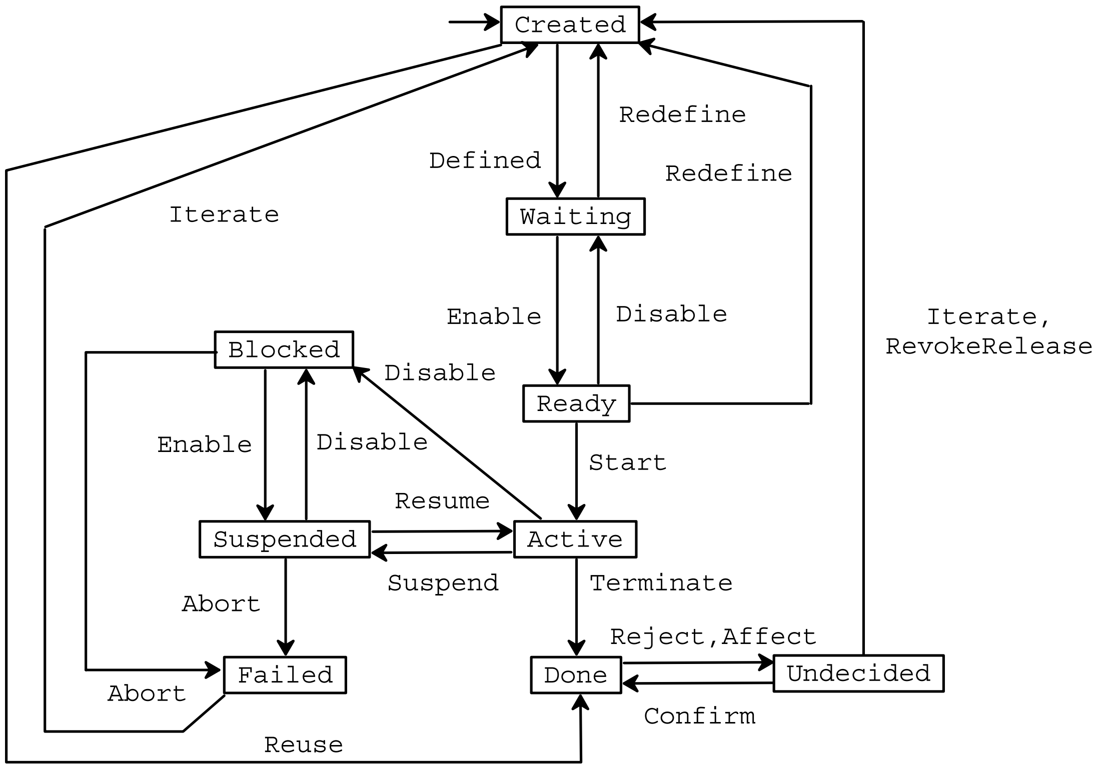

Figure 12 State transition diagram for processes

Furthermore, we have to consider the impact of a state change on dependent processes. A rigorous solution would enforce the termination condition as an invariant, i.e. if a master process leaves state `Done`, all terminated dependent processes would have to leave state `Done`, as well. However, such a restrictive rule implies a ripple effect of a state change: all (directly or transitively) dependent processes would have to leave state `Done`, too. This is particularly annoying if state changes are accompanied by notifications and are required to be undone manually. In general, a state change will only affect a certain fraction of dependent processes. Therefore, rippling state changes imply that lots of redundant messages are sent, and lots of manual repair actions are required.

In order to avoid ripple effects, we have adopted the following approach to change propagation:

- A transition from Done to Undecided does not affect dependent processes.
- When a process moves from Undecided to Created (the initial state of the diagram), all dependent processes in state Done are forced into state Undecided.
- If any master process revokes its release for a dependent process in state Undecided, the latter is forced into state Created.
- Creation or deletion of incoming data flows of a process in state Done or state Undecided is prohibited. Since these changes affect the process definition, they will eventually cause a reactivation. Therefore, the process must return to state Created (via Undecided) to perform such a drastic change.

In this way, we make sure that changes are propagated step by step as required. This is illustrated in Figure 11 where processes are denoted by p_1, p_2, ... and ✓, ?, a represent states Done, Undecided, and Active, respectively.

To conclude this section, let us discuss the *state transition diagram* shown in Figure 12. The State attribute is attached to nodes of type ComponentProcess which is a subtype of VersionComponent. The states have the following meaning:

- Created serves as initial state where the process is defined. Incoming data flows are created in order to define its inputs. The process has a single output, namely some version of the object to which the component refers. If no version number is specified, the output is created from scratch (development process). Otherwise, the specified version serves as starting point for the output to be created (maintenance or enhancement process). In order to leave Created, incoming data flows must be complete with respect to the schema. In case of the Defined transition, an actor must have been made responsible for executing the process. Reuse indicates that the specified version may be reused without modification.
- In state Waiting, the process waits for its input data. Redefine returns to Created. Enable is performed automatically as soon as the activation condition is fulfilled.
- In state Ready, the process waits for being started by the actor (transition Start). Disable is performed automatically when the activation condition is violated. Redefine returns to Created.
- Active is a persistent state which may last for minutes, hours, days, ... depending on the complexity of the process to be carried out. Process execution may involve multiple work sessions and multiple tool activations. Multiple versions may be developed and pre–released to dependent processes one after the other. Ideally, completeness and consistency of the output increase monotonically, which means that the release set is gradually extended and more and more propagation attributes of outgoing data flows are assigned true. However, pre–releases and propagations may also be revoked. Disable is performed automatically when the activation condition is violated. In contrast to this, Suspend has to be invoked explicitly e.g. when the actor discovers errors in inputs. Terminate indicates successful termination of the process.

- Blocked and Suspended mean that process execution has been interrupted. In particular, both states are used to handle feedbacks. Blocked and Suspended differ in that the activation condition is violated/fulfilled, respectively. In both states, the process may revoke pre–releases of its outputs; furthermore, the process definition may be changed. Resume returns to state Active, and Abort indicates failure.
- Failed means that the process has failed to achieve its goals. Either the process is reactivated later on (transition Iterate), or it is removed from the net because its result is not needed any more.
- Done indicates successful process termination. Later on, the state Done may have to be left for different reasons. The transition Reject is invoked in case of feedback from a dependent process. Affect indicates that inputs have changed and the process may have to be reactivated.
- Finally, Undecided serves as intermediate state for processes which may have to be reactivated. Confirm, which is successful under the same conditions as Terminate, returns to state Done. On the other hand, Iterate and RevokeRelease process feedbacks from dependent processes and significant changes to inputs, respectively.

	Created	Waiting	Ready	Active	Blocked	Suspended	Done	Undecided
Define Process	+	–	–	–	+	+	–	–
Execute Process	–	–	–	+	(–)	(–)	–	–
Activation Condition	*	–	+	+	–	+	+	+
Termination Condition	*	*	*	*	*	*	(+)	*
Stability	+	+	+	*	*	*	+	+
Consistency	*	*	*	*	*	*	+	+

Table 1 Characterization of process states

In order to further clarify the semantics of states, Table 1 summarizes some important *state features*. States are characterized by admissible operations and values of boolean attributes. In case of operations, entries + and – indicate that the respective operation is permitted or prohibited, respectively; in case of boolean attributes, they denote the values true and false, respectively (* denotes any value). The rows of the table have the following meaning:

- Define Process: creation/deletion of incoming data flows, definition of a starting version (if any), and assignment of an actor,
- Execute Process: derivation of a new version, modification of a version, replacement of the current version with another one, freezing/thawing, releasing and revoking of the current

version (note that in states `Blocked` and `Suspended` pre–releases may be revoked, but no other operations of the category `DefineProcess` are applicable),

- `Activation Condition`: the condition under which the process may be started, resumed, or confirmed,
- `Termination Condition`: the condition under which the process may be terminated (note that the entry + for state `Done` is enclosed in parentheses because the termination condition must hold for `Terminate`, but transitions of master processes from `Done` to `Undecided` are allowed later on),
- `Stability`: indicates whether the current version produced by the component process is frozen or may be modified,
- `Consistency`: indicates whether the current version is consistent with respect to the enclosing configuration.

THE SUKITS PROTOTYPE

Within the SUKITS project, we built a prototype of the CIM Manager which was completed in fall 1993 and is currently being redesigned and extended. In order to evaluate the prototype, two sample scenarios were investigated, namely development of *single metal parts* and *plastic parts* produced by injection moulding, respectively. Both scenarios are rather complex; the corresponding schemas both comprise about 15 document types and 30 dependency types. Documents to be managed include e.g. CAD designs for different purposes (e.g. geometries of the single part to be produced, of the raw part to start with, of the positions of the working piece, etc.), manufacturing plans (CAP), NC programs (CAM), material requirements definitions, results of FEM simulations (CAE), etc. The CIM Manager was tailored to both scenarios by defining concrete models and integrating application systems.

In both scenarios, we succeeded in managing engineering products and processes in a tightly integrated way. For each scenario, a demonstration was worked out which illustrates net extensions, pre–releases, and feedbacks. The demonstrations showed the feasibility of our approach. In particular, product–centered process management proved to be both powerful and conceptually simple.

The *architecture* of the SUKITS prototype is shown in Figure 13. Basically, we follow a client–server paradigm with CIM application systems residing on client machines and management tools provided by the server machine. However, an application system may also be run on the server machine, and we plan to extend the prototype so that the server may be distributed across multiple machines.

The components residing on the server machine have been implemented with the help of services and tools which were developed in the IPSEN project, a research project which is devoted to the design and implementation of tightly integrated, structure–oriented software development environments (Nagl, 1990). Here, we have heavily reused software which was originally dedicated to a different application domain, namely software engineering. The integration database is realized on

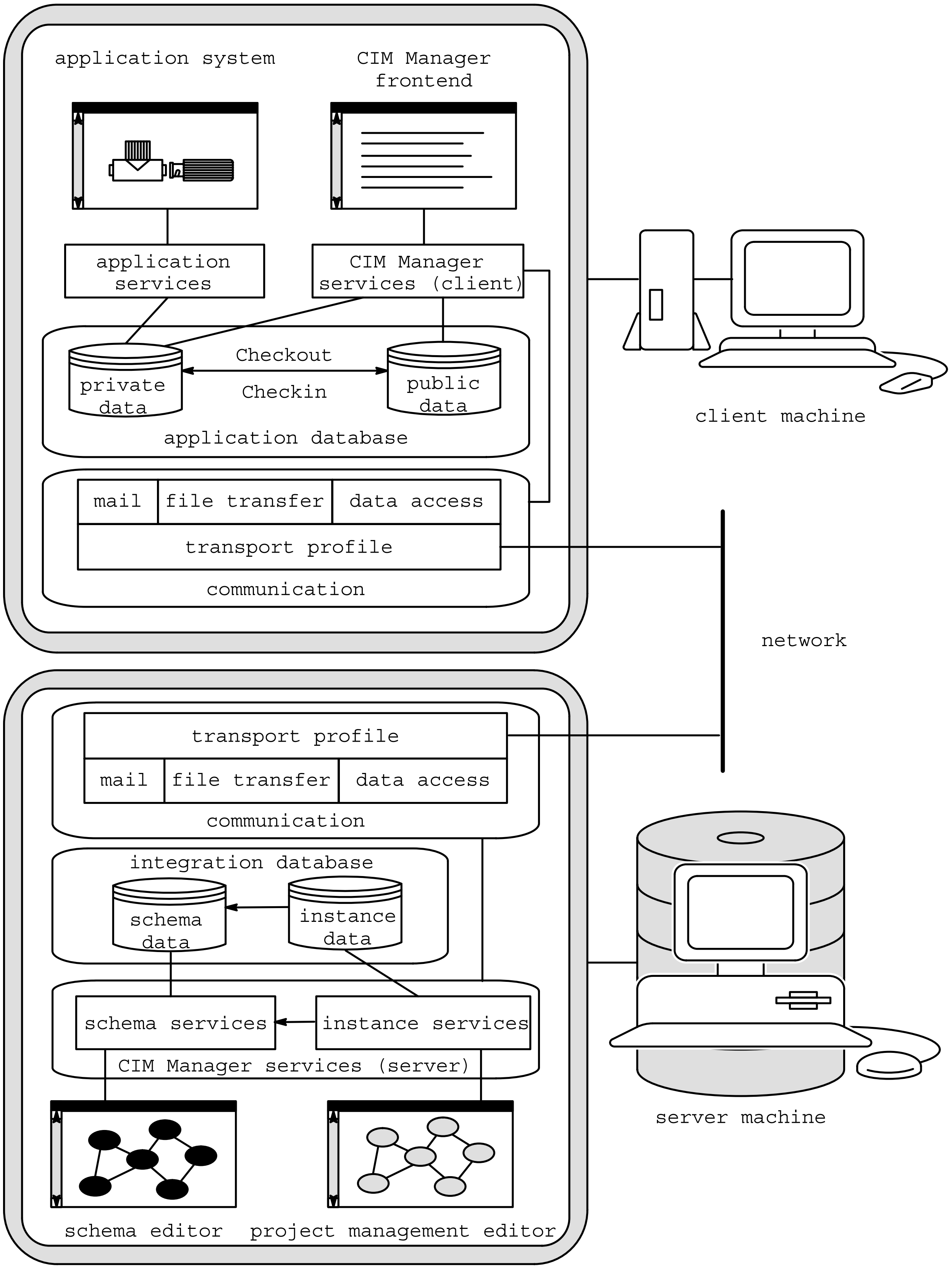

Figure 13 Architecture of the SUKITS prototype

top of the GRAS database system (Kiesel, Schürr, and Westfechtel, 1995) which supports storage and manipulation of attributed graphs and provides a lot of useful features such as undo/redo, nested transactions, and client/server distribution (the latter of which, however, is restricted to a homogeneous network). A layer of services used for operating on schema and instance data is placed on top of GRAS. Structure–oriented, graphical tools provide a comfortable user interface to these services. The *schema editor* is used to tailor product and process management services to a certain application domain. The *project management editor* is employed to edit configurations and assign tasks to developers.

On the client machine, a *front end* (which was implemented with the help of a commercial user interface tool kit) supports developers with a set of basic functions needed for the execution of atomic development processes (i.e. processes which are not structured any further). In particular, revisions of documents are checked into a private workspace where they are manipulated by means of application systems; whenever appropriate, intermediate or final revisions are checked back into the public workspace. Furthermore, application systems are started via the front end.

All components of the SUKITS prototype are glued together by an *OSI–based communication system* providing for mail, data access, and file transfer in a heterogeneous network (Hermanns, 1993; Hermanns and Engbrocks, 1994). The whole prototype comprises more than 60,000 lines of code written in Modula–2 and C (plus 250,000 lines of code of the IPSEN environment).

RELATED WORK

We have presented a system which manages engineering design products and processes in an integrated way. The CIM Manager manages versioned, interdependent documents which are combined into configurations. These configurations serve as workspaces for integrating activities and results which are carried out by multiple engineers. Management of engineering design processes supports net extensions, pre–releases of results, and feedbacks. To sum up, we have built a framework for integrated product and process management which is both powerful and conceptually simple. Although the SUKITS project addresses mechanical engineering as its main application domain, our approach is by and large domain–independent, i.e. it can be applied to other domains such as electrical engineering, software engineering, or office automation.

The CIM Manager may be classified as a *CAD framework* (Harrison et al., 1990), i.e. an infrastructure for managing engineering products, processes, and resources. Typically, a CAD framework supports a posteriori integration of application systems. However, some CAD frameworks provide for different kinds of integration, depending on the way framework services are used by application systems. In particular, applications which are developed on top of the framework can be integrated more closely. Typical examples for CAD frameworks are the CadLab system (Brielmann et al., 1992) and the Nelsis CAD framework (van der Wolf, Bingley, and Dewilde, 1990). Note that infrastructures with similar goals and properties have also been developed in other domains, e.g. PCTE (Wakeman and Jowett, 1993) for software engineering.

Research in product management – which has also been called version management, configuration management, or engineering data management – has been carried out in multiple domains (Dart, 1992). A survey on the CAD domain is provided in (Katz, 1990), approaches in the software engineering domain are described in (Whitgift, 1991). During the last years, a lot of commercial tools have entered the market. Although the most essential modeling concepts seem to have been identified, integration of these concepts is still an open issue. Our approach to product management is an effort to integrate at least some essential concepts into a coherent framework (versioning, dependencies, complex objects).

Similar approaches have been realized e.g. in the Nelsis CAD framework, in the CadLab system, in OVM (Käfer and Schöning, 1992), or in the ATIS class hierarchy underlying the DEC Cohesion Environment (Welsh, 1991). Among these, OVM is the only one which distinguishes between an object plane and a version plane in a way similar to our approach. However, configurations are modeled as sets of versions rather than as graphs containing applied occurrences of versions. Furthermore, configurations cannot be put under version control. In our model, configurations play a very important role not only because they define complex products, but also because they act as workspaces for integrating engineering activities. Furthermore, the distinction between definitions and applied occurrences of versions is crucial because context–dependent information (e.g. configuration consistency) must be attached to version components in configurations. Among the approaches cited above, only CadLab supports this distinction. However, the CadLab model is tailored towards CAD applications, in particular to representation of design hierarchies, while we have strived for keeping the model domain–independent.

Management of engineering design processes may be supported in different ways. *Process control* is concerned with work assignments, scheduling of activities, monitoring, etc. This is the kind of support which is required by project managers. *Process recording* captures the actual design process in order to document design decisions. In this way, the impacts of changes can be assessed more accurately because detailed information about the design rationale is available. So far, we have been focusing on process control and have not studied process recording (Lee and Lai, 1991).

Approaches to process control may be divided into two groups. *Product–centered process control*, which is realized in the CIM Manager, enriches the product structure with process–related information. *Separate process control* divides the process structure from the product structure. Both approaches have their advantages and weaknesses. We have favoured product–centered process control because it is conceptually simpler. The user only has to deal with one structure instead of two, namely the product structure and the process structure which are closely related and must be kept consistent with each other. On the other hand, separate process control is more complex to handle, but also more general. Although frequently there is a close correspondence between process structure and product structure, some kinds kinds of processes cannot be represented in our product–centered approach in a satisfactory way.

Let us briefly review some related product–centered approaches to process control: In Adele (Belkhatir, Estublier, and Melo, 1994), processes are described by means of events and associated actions. Events constitute a base mechanism which can be used e.g. to implement the process control approach described in this paper. In (Tankoano, Derniame, and Kaba, 1994), a frame–based knowledge representation language is used for process control. Software processes are modeled by facets which, among other things, describe ordering constraints. Finally, in (Humphrey and Kellner, 1989) a product–centered approach is described which is based on state charts. To each document, a state chart is attached. State transitions may be coupled across different state charts. However, dependencies between documents are not modeled explicitly. Furthermore, the approach cannot handle dynamic instantiation of documents (and corresponding processes) because the state charts to be coupled must be known at specification time.

Some approaches which separate product and process structure are based on hierarchical Petri nets. However, these approaches have not yet provided satisfactory solutions for the problem of dynamic process evolution. In SPADE (Bandinelli, Fuggetta, and Ghezzi, 1993) and PROCESS WEAVER (Fernström, 1993), the structure of a subnet in the hierarchy cannot be modified after execution of the subnet has started. MELMAC (Deiters and Gruhn, 1990) does allow for dynamic structural modifications, but treats them as an exception rather than the rule. All approaches have in common that they do not distinguish between type and instance level. In this respect, our approach is close to EPOS (Jaccheri and Conradi, 1993) which manages nets of task which are dynamically instantiated from task types. However, the execution semantics of EPOS tasks is much simpler and less powerful than ours; in particular, it does not take concurrent engineering into account.

To conclude this section, let us compare our work with other approaches to concurrent engineering: Concurrent engineering is still a young field. Currently, many different approaches to concurrent engineering are being elaborated, but evaluation and consolidation still need to be carried out. It is still an open problem how much concurrency is actually desirable, and in which way concurrently executed processes should be coordinated. The approach presented in this paper tries to achieve a balance between sequential, waterfall–like processes and chaotic concurrent processes where subprocesses may start before their goals and inputs have been defined. This balance is essentially achieved by sophisticated management of pre–releases. Other approaches allow for more concurrency, but take the risk of increasing rather than reducing the amount of work for executing a software project. For example, in (Pohl and Jacobs, 1994) a three phase model is proposed where documents in different work areas are developed concurrently and independently in the first phase and are integrated later on. A posteriori integration is difficult and expensive; early coordination would save much effort. In (Pulli and Heikkinen, 1993), processes exchange intermediary results like in our approach; furthermore, they can initially rely on assumptions with respects to results of master processes. Since such implicit assumption may prove false, we believe that engineers should communicate to validate their assumptions. Such a communication may again result in pre–releases, or it may be handled in an informal way (e.g. by electronic mail).

CONCLUSION

We have presented an integrated approach to product and process management. We have implemented a prototype which demonstrates the feasibility of this approach. Notwithstanding, there are many open problems which still have to be studied and solved. Our long–term goal is to develop a reactive framework for supporting the development of complex technical products. A vision of such a framework is presented in (Nagl and Westfechtel, 1994). This framework goes far beyond the current SUKITS prototype in various respects. So far, we have studied evolution of process nets which is controlled by an ER–like schema. However, we only have a limited understanding of how administrative processes controlling net evolution should be modeled. Currently, any operation can be applied in any state, provided that its does not violate the constraints imposed by the meta model and the concrete model. Furthermore, we have not yet addressed schema evolution during project run–time. We also have to study cooperation between different projects, both within an enterprise and across different enterprises. Finally, integration has to be extended to the fine–grained level in order to provide for incremental change propagation which is essential for a reactive environment.

ACKNOWLEDGEMENTS

This work was partially supported by the German Research Council (DFG) under the project title SUKITS (Software and Communication Structures in Technical Systems, grant Na 134/5). The author is indebted to all members of the SUKITS project: Frank Boshof, Alexander Pollack, Sebastian Repetzki, Karsten Sonnenschein, Martin Walz (all WZL), Dirk Menzenbach (IKV), Peter Heimann, Joachim Schwartz (Lehrstuhl für Informatik III), and Oliver Hermanns (Lehrstuhl für Informatik IV).

REFERENCES

Bandinelli, S., Fuggetta, A., and Ghezzi, C. (1993) “Software Process Model Evolution in the SPADE Environment,” *IEEE Trans. Software Eng.* 19(12): 1128–1144.

Belkhatir, N., Estublier, J., and Melo, W. (1994) “ADELE–TEMPO: An Environment to Support Process Modelling and Enaction,” in A. Finkelstein, J. Kramer, and B. Nuseibeh, Eds., *Software Process Modelling and Technology*, John Wiley & Sons, New York, pp. 187–222.

Brielmann, M., Kupitz, E., Mallon, D., Stewing, F.–J., and Weißenberg, N. (1992) “A Common Data Schema for Tool Integration,” *Proc. CAD ’92*, Springer–Verlag, Berlin, pp. 127–140.

Dart, S.A. (1992) “Parallels in Computer–aided Design Frameworks and Software Development Environments Efforts,” Third Workshop Electronic Design Automation Frameworks, *IFIP Transactions* A–16: 175–189.

Deiters, W. and Gruhn, V. (1990) “Managing Software Processes in MELMAC,” Proc. 4th Symp. Software Development Environments, *ACM Software Eng. Notes* 15(6): 193-205.

Eversheim, W., Weck, M., Michaeli, W., Nagl, M., and Spaniol, O. (1992) “The SUKITS Project: An approach to a posteriori Integration of CIM Components,” *Proc. GI–Jahrestagung ’92*, Informatik aktuell, Springer–Verlag, Berlin, pp. 494–503.

Fernström, C. (1993) “PROCESS WEAVER: Adding Process Support to UNIX,” *Proc. 2nd Int. Conf. Software Process*, IEEE, Los Alamitos, pp. 12-26.

Harrison, D., Newton, A., Spickelmeir, R., and Barnes, T. (1990) “Electronic CAD Frameworks,” *Proc. of the IEEE* 78(2): 393–419.

Hermanns, O. (1993) “Data Access Protocols for Integrated Engineering Environments”, *Proc. COMPEURO ’93: Computers in Design, Manufacturing and Production*, IEEE, Los Alamitos, pp. 350–357.

Hermanns, O. and Engbrocks, A. (1994) “Design, Implementation and Evaluation of a Distributed File Service for Collaborative Engineering Environments,” *Proc. Third Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, IEEE, Los Alamitos, pp. 170–175.

Humphrey, W. and Kellner, M. (1989) “Software Process Modeling: Principles of Entity Process Models,” *Proc. 11th Int. Conf. on Software Eng.*, IEEE, Los Alamitos, pp. 331–342.

Jaccheri, M. and Conradi, R. (1993) “Techniques for Process Model Evolution in EPOS,” *IEEE Trans. Software Eng.* 19(12): 1145–1156.

Käfer, W. and Schöning, H. (1992) “Mapping a Version Model to a Complex Object Data Model,” *Proc. 8th Int. Conf. Data Eng.*, IEEE, Los Alamitos, pp. 348–357.

Katz, R.H. (1990) “Toward a Unified Framework for Version Modeling in Engineering Databases,” *ACM Computing Surveys* 22(4): 375–408.

Kiesel, N., Schürr, A., and Westfechtel, B. (1995) “GRAS, a Graph-Oriented (Software) Engineering Database System,” *Information Systems* 20(1): 21–51.

Kosanke, K. (1991) “The European Approach for an Open System Architecture for CIM (CIM–OSA) – ESPRIT project 5288 AMICE,” *Computing & Control Eng. Journal* 2(3): 103–109.

Lee, J. and Lai, K.–Y. (1991) “What’s in a Design Rationale?,” *Human–Computer Interaction* 6(3/4): 250–280.

Nagl, M. (1990) “Characterization of the IPSEN Project,” in N. Madhavji, W. Schäfer, and H. Weber, Eds., *Proc. 1st Int. Conf. on Systems Development Environments & Factories*, Pitman Press, London, pp. 141–150.

Nagl, M. and Westfechtel, B. (1994) “A Universal Component for the Administration in Distributed and Integrated Development Environments,” Technical Report AIB 94–8, Technical University of Aachen, Germany.

Pohl, K. and Jacobs, S. (1994) “Concurrent Engineering: Enabling Traceability and Mutual Understanding,” *Concurrent Eng.: Research & Applications* 2(4): 279–290.

Pulli, P. and Heikkinen, M. (1993) “Concurrent Engineering for Real–Time Systems,” *IEEE Software* 10(11): 39–44.

Reddy, R., Srinivas, K., and Jagannathan, V. (1993) “Computer Support for Concurrent Engineering,” *IEEE Computer* 26(1): 12–16.

Schwartz, J. and Westfechtel, B. (1993) “Integrated Data Management in a Heterogenous CIM Environment,” *Proc. COMPEURO ’93: Computers in Design, Manufacturing and Production*, IEEE, Los Alamitos, pp. 248–257.

Tankoano, J., Derniame, J., and Kaba, A. (1994) “Software Process Design Based on Products and the Object Oriented Paradigm,” *Proc. Third European Workshop Software Process Technology*, LNCS 772, Springer–Verlag, Berlin, pp. 177–185.

van der Wolf, P., Bingley, P., and Dewilde, P. (1990) “On the Architecture of a CAD Framework: The Nelsis Approach,” *Proc. 1st European Design Automation Conf.*, IEEE, Los Alamitos, pp. 29–33.

Wakeman, L. and Jowett, J. (1993) *PCTE – The Standard for Open Repositories*, Prentice Hall, New York.

Welsh, T. (1991) “Digital’s COHESION Environment,” in F. Long, Ed., *Software Engineering Environments, vol.3*, Ellis Horwood, London, pp. 363–378.

Westfechtel, B. (1995) “Using Programmed Graph Rewriting for the Formal Specification of a Configuration Management System,” *Proc. 20th Workshop on Graph–Theoretical Concepts in Computer Science WG ’94*, LNCS, Springer–Verlag, Berlin.

Whitgift, D. (1991) *Methods and Tools for Software Configuration Management*, John Wiley & Sons, New York.