in: Proc. Int. Conf. on Data Engineering ‘94, Houston, Texas, 1994, pp. 440-451.

Capturing Design Dynamics - The CONCORD Approach

N. Ritter, B. Mitschang, T. Härder, M. Gesmann, H. Schöning
University of Kaiserslautern, CS Department
P.O.Box 3049, 67653 Kaiserslautern, Germany
e-mail: {ritter | mitsch | haerder | gesmann | schoenin}@informatik.uni-kl.de

Abstract

‘Computer-Supported Cooperative Work’ is a young research area considering applications with strong demands on database technology. Especially design applications need support for cooperation and some means for controlling their inherent dynamics. However, today’s CAD systems mostly consisting of a collection of diverse design tools typically do not support these requirements. Therefore, an encompassing processing model is needed that covers the overall design process in general as well as CAD-tool application in particular. As a consequence, this model has to be rich enough to reflect the major characteristics of design processes, e.g., goal-orientation, hierarchical refinement, stepwise improvement as well as team-orientation and cooperation. The CONCORD model that will be described in this paper, reflects the distinct properties of design process dynamics by distinguishing three levels of abstraction. The highest level supports application-specific cooperation control and design process administration, the second considers goal-oriented tool invocation and work-flow management while the third level provides tool processing of design data. To achieve level-spanning control, we rely on transactional facilities provided at the various system layers.

1. Introduction & Overview

Facing the growing complexity of technical products, the process of design is typically carried out by a team of cooperating designers rather than by a single person. Several methodologies have been developed to structure the overall design process and to support designers working on partial design problems and cooperating with each other, e.g., by negotiating their individual design goals or by exchanging their partial results. However, today’s CAD systems typically do not support cooperative work in a satisfactory manner. Exchange of preliminary results is usually done without system support and control. In larger design teams this causes inconsistencies in design objects which must be resolved by hand with a considerable overhead. In our opinion, such problems can be faced by extending database technology with ‘cooperation capabilities’.

1.1 Computer-Supported Cooperative Work and Database Technology

Computer-Supported Cooperative Work [SB92, RMB92] is a young field of research which attracted much attention in the last few years, especially in the design area. Its goal is to provide a conceptual framework that supports the requirements of cooperative work arrangements. In the following, we will briefly discuss some major characteristics of cooperative applications (also mentioned in [RMB92] and [RSJK93]), thus distinguishing them from other kinds of computer applications.

(1) Distribution: Ideally a complex design process is partitioned into a set of tasks to be carried out by a geographically dispersed team of designers, who use a computer-supported environment for collaboration. That network-based environment comprises design tools and design data repository as well as communication facilities.

(2) Coordination: On one hand, the designers work in parallel, each on his individual task. On the other hand, effective communication facilities should enable the designers to cooperate in order to produce a high quality product within a shorter turnaround time (concurrent engineering). These facilities have to support team decision-making as well as negotiation concerning design goals and solutions between the geographically scattered team members.

(3) Information sharing: Obviously, the foundation of cooperation is sharing of information. Information about the design process (e.g., constraints across multiple perspectives) and information generated during the design process (e.g., data derived by design tools) need to be managed. A major issue is consistency maintenance during concurrent change of the shared design information stored in common data repositories.

Managing shared information has ever been in the realm of database systems and the database research community has traditionally tackled new problems arising in that area. However, as discussed in [GS87a], cooperative applications are placing a challenging and novel set of demands on database technology. These involve, among others, the development of adequate data representations, version control mechanisms, activity management facilities, and concurrency control mechanisms. Further requirements arise while coping with distributed and heterogeneous databases. Instead of addressing the whole problem area, in this paper we will focus on the dynamic issues of cooperative applications, thereby abstracting from specific design data representation and management via repositories. The main questions in the context of design dynamics are:

- How is data supply (checkout, checkin) and data processing (preservation of reference locality) achieved for design tools?
- How can the ‘work flow’ of design (tool) applications be (pre-)planned and scheduled in order to apply certain design methodologies?
- How can cooperation in concurrent engineering be controlled in order to keep the design (i.e., the subject of cooperation) consistent esp. in view of possible ‘failure’ cases?

It is commonly known that the traditional ACID paradigm (atomicity, consistency, isolation, durability) of conventional transactions [HR83] developed for small units of work accessing only few data items and with a short system residence time is not applicable in cooperative work arrangements. Serializability as the notion of correctness is too restrictive. The isolation property builds ‘protective walls’ among concurrent transactions and is therefore contrary to cooperation. Furthermore, the atomicity property is not adequate for long duration activities as, for example, the application of design tools. As a consequence, extended transaction models have been proposed to support, among others, cooperative applications.

1.2 Previous Work in Advanced Transaction Models

We can find a lot of approaches in the literature proposing advanced transactional models that try to support the dynamics of new database applications. The first extensions to flat ACID transactions were not proposed to support cooperative work specifically. However, they introduced some basic concepts which are also useful in cooperative work arrangements. For example, the model of ‘Nested Transactions’ [Mo81] allows with its non-vital subtransactions for fine-granuled units of recovery and for the use of subtransactions for contingency purposes. The ‘Sagas’ model [GS87b] is based on one hand on the idea to release resources as soon as possible and on the other hand on the concept of compensating transactions allowing for semantic undo operations of already ‘committed’ transactions. These basic ideas are also part of more recent proposals that were (especially) targeted to reflect requirements of cooperative work [El92]. Most of these approaches rely on (sub-)***transaction hierarchies*** allowing for a natural mapping of real cooperative units of work to a number of interdependent database transactions (‘Cooperative Transaction Hierarchies’ [NRZ92], “Cooperative SEE Transactions’ [HHZB92], ‘Flex Transactions’ [KPE92], ‘Tool Kit Transactions’ [US92], ‘Multi Level Transactions’ [WS92]). On the other hand, ‘Split Transactions’ [PKH88] and ‘ConTracts’ [WR92] support long-lived activities built upon only flat transaction structures. The basic idea of ConTracts is to model ***control flow*** between predefined actions (called steps) which can be combined to atomic units (ACID transactions). The modeling of control flow is a means to achieve recoverability of design states where the loss of work is minimized. This means that in the case of a failure, the actual context can be reestablished and the execution can be continued. Furthermore, ConTracts allow to externalize partial result, thereby relying on invariants for concurrency control. This aspect leads us to the notion of ***transaction correctness*** appropriate in cooperation environments. While serializability is too restrictive, some of the considered models introduce concepts to enable the user to specify the correctness criteria (‘Cooperative Transaction Hierarchies’ [NRZ92], ‘Cooperative SEE Transactions’ [HHZB92], ‘Tool Kit Transactions’ [US92]). These approaches have in common that with every node in the transaction hierarchy a local database is associated. Special mechanisms are attached to this so-called *object pool* in order to control the concurrent work of the subtransactions that are associated with that node. The ‘Tool Kit’ approach allows the user to build a ‘heterogeneous’ transaction tree with different types of subtransactions provided by an application-specific transaction manager. On the other hand, the ‘Cooperative Transaction Hierarchies’ model allows the user to specify the local correctness of a node by means of ‘patterns and conflicts’ [Sk91], which basically restrict the possible sequences of actions on the node’s object pool.

Our approach is different from those considered so far. We do not want to propose (yet) another advanced transaction model. In the contrary, we built upon approved transaction concepts. Our CONCORD model *(**Con**trolling **Co**ope**R**ation in **D**esign Environments)*, which will be discussed in this paper, provides a framework of generic facilities allowing for flexible management of the design process and for a controlled collaboration between designers. The approach is targeted towards the primary subjects of design. Sect.2 will give a brief overview of the CONCORD model. Due to the diversity of the properties to support, we perceived a layered approach, where each abstraction layer provides a certain set of concepts inherent to design dynamics. After introducing a particular approach to VLSI design as our sample design process (Sect.3), we will detail in Sect.4 on the concepts provided at each level. In Sect.5, we introduce the activity managers that control a cooperative design process. In addition to the discussion of their services, we outline a global failure model and identify the necessary DBMS support. The last section gives a conclusion and an outlook to further work.

2. Overview of the CONCORD Model

The CONCORD model captures the dynamics inherent to design processes. It is developed in a top-down fashion starting from the intricacies of design applications in order to meet their inherent requirements, such as:

- *hierarchical refinement* and *decomposition* of the design tasks and the corresponding design objects,
- *goal orientation* of each design task as unit of work in the design process,
- *stepwise improvement* of preliminary design states,
- *team orientation* and *cooperation* among design tasks,
- *design-specific consistency* of design tasks and design states.

To reflect these different requirements, the CONCORD model distinguishes three different levels of abstraction illustrated in Fig.1.

Administration/Cooperation Level (AC level)

At the highest level of abstraction, we consider the more creative and administrative part of design work. There, the focus is on the description and delegation of design tasks as well as on a controlled cooperation among the design tasks. The central concept at this level is the ***design activity*** (***DA***). A DA is the operational unit representing a particular design task or subtask. During the design process, a ***DA hierarchy*** can be dynamically constituted resembling (a hierarchy of) concurrently active tasks. All relationships between DAs are explicitly modeled, thus capturing design flow (cooperation relationship *delegation*), exchange of design data (cooperation relationship *usage*), and negotiation of design goals (cooperation relationship *negotiation*). The inherent integrity constraints and semantics of these cooperation relationships are enforced by a central system component, called the ***cooperation manager.***

Design Control Level (DC level)

Looking inside a DA reveals the *DC level*. There, the organization of the particular actions performed in order to fulfill a certain (partial) design task is the subject of consideration (***work flow***). Fig.1 shows at this level an execution plan (***script***) of a particular design activity. This script models the ***control/data flow*** between several design tool executions. The operational unit serving for the execution of a design tool is the ***design operation*** (***DOP***). In order to control the actions according to the scope of one DA, but without restricting the designers' creativity, flexible mechanisms for specifying the work flow for a DA (scripts, constraints, event-condition-action rules) are provided. The correctness of tool executions is guaranteed by a system component, called ***design manager***. The design manager does also provide for recoverable script executions that is needed for level-specific and isolated failure handling as discussed in more detail in Sect.5. Design tools are applied to improve existing design states in order to finally reach at a design state that completes the current (partial) design task. Design states are captured by means of a version model managed by the design data repository. The derivation of design states, i.e. ***design object versions*** (***DOV***), by means of tool applications is supported by concepts provided at the TE level.

Tool Execution Level (TE level)

From the viewpoint of the DBMS or data repository, a DOP is an ACID transaction. Due to long duration, it is internally structured by save/restore and suspend/resume facilities as illustrated in Fig.1. A DOP processes design object versions in three steps. First, the input versions are checked out from the integrated data repository. Second, the loaded object data is processed by the design tool. Third, the finally derived new version is propagated back to the data repository (checkin operation). The derivation of schema-consistent and persistent design object versions is guaranteed, again, by a central system component, called ***transaction manager***. It is also responsible for the isolated execution of DOPs and for recoverable DOP executions that are, again, necessary for a level-specific and isolated failure handling. The transaction manager employs mechanisms provided by the advanced DBMS which manages the integrated data repository.

The brief overview given above shows that we are going on one hand a similar way as the ConTracts approach: we claim that even in cooperative design applications there are still units of work to be processed atomically (within the larger activities of a designer), which need to be organized by work-flow capabilities. These atomic units are encapsulated as sequences of elementary operations and their intermediate results need not to be seen by other designers. Therefore, ACID transaction (at TE level) and work-flow capabilities (at DC level) are an integral part of our model. However, the cooperation aspect is missing in ConTracts. For this reason, we propose an additional layer to reflect the conviction that cooperation takes place on a higher abstractional niveau (AC level). To embed the semantics of cooperation, the CONCORD model provides a number of generic facilities which allow for modeling and managing the design process. Before discussing the modeling concepts in more detail we will look into a particular and practically approved approach to VLSI design [Zi86] as our sample design process scenario.

3. A Methodology to VLSI Design

Electronic design is a CAD application area that is well-known for its high demands for effective data management facilities as well as for adequate support for design management and design methodologies. In order to deal with the ever increasing complexity of the design process, the design methodology described in [Zi86] distinguishes four different design domains as depicted in Fig.2.

The domain ***behavior*** contains the functional specification (e.g. algorithmic description) of the circuit to be designed, whereas the domain ***structure*** describes the composition of the design object in an abstract (realization independent) manner. The aspects of the physical design are concentrated in the two remaining domains. In the domain ***floorplan*** the topography of the circuit is considered, which is refined to the physical realization in the domain ***mask layout***. The second dimension of the design plane of Fig.2 is given by the design object hierarchy that groups design objects at different levels. A sample four-level cell hierarchy is sketched on the right-hand side of Fig.2. In this scenario

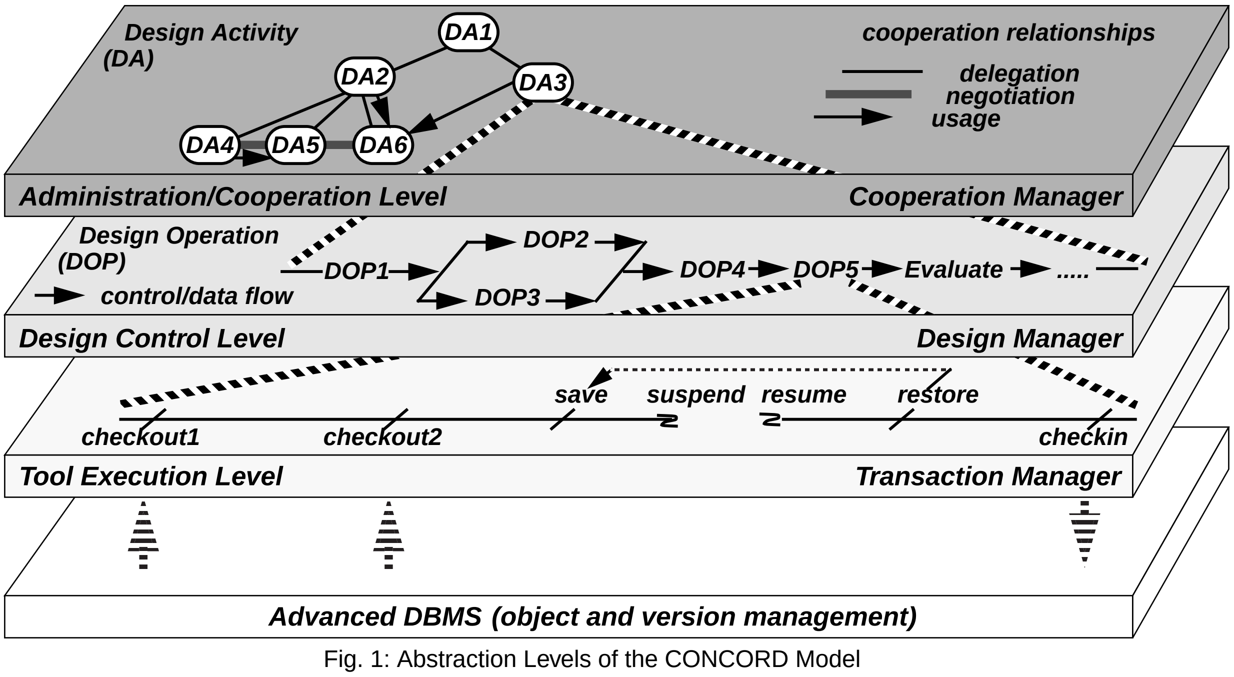


Fig. 1: Abstraction Levels of the CONCORD Model

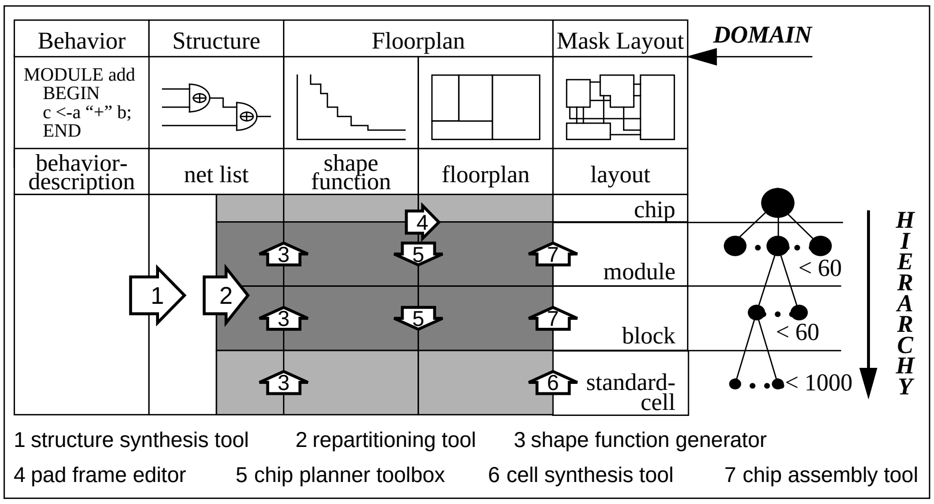


Fig.2: Design Plane

a chip is divided into modules representing arithmetic-logic unit, control unit, and so on; each module, in turn, can be partitioned into blocks at the next level (e.g., read-only memory, instruction decode, etc.) and each of these blocks is again partitioned into standard cells at the lowest level (e.g., multiplexer, AND-circuit, etc.). The arrows of Fig.2 illustrate that the design process starts with a behavioral description of the circuit to be designed and then traverses the design plane from left to right. The design is carried out by application of design tools, thereby enhancing and completing previous work. In Fig.2, each arrow is assigned to a particular design tool by an associated number.

For simplicity purposes, we will only focus on the chip planning phase of the VLSI design process (toolbox 5 in Fig.2). Chip planning is a creative and cooperative process which in most cases cannot be done without designer interaction and which proceeds as follows. In a top-down fashion, a floorplan is computed for each cell of the hierarchy by recursively applying the chip planner. These computations are based on estimated information about its subcells (i.e., shape functions indicating the possible shapes of the subcells provided by tool 3 in Fig.2). Further information about the CUD (cell under design) and its subcells, e.g., the connections of the subcells, is decoded in the module and net list (cf. Fig.3). The most important input is the interface description of the CUD, expressing non-functional requirements as, for example, the shape of the CUD and the positions of the pin intervals on the CUD's frame. As mentioned previously, the chip planner is a tool box containing several tools: *bipartitioning*, *sizing*, *dimensioning*, and *global routing*. Due to space restrictions we will not give a detailed description of the internal processing, but it is important to know, that the designer may perform reiterations of parts of the internal tool executions in order to achieve optimal space exploitation. As a result, the chip planner arranges the subcells and the connecting channels for wiring within the given area of the CUD (floorplan). This includes the interface descriptions of the subcells which are input data for subordinate planning steps.

Obviously, one can recognize that all important characteristics of the VLSI design process are reflected in particular concepts provided by the CONCORD model. Design task delegation and design task hierarchy (AC level) can be derived from the design plane depicted in Fig.2, i.e. from the partitioning of the design process into phases, or from the application of particular design tools (indicated by the arrows in Fig.2) as well as from the complex structure of the VLSI cell hierarchy. In Sect.4 we will give an example. The DC level is explicitly given by Fig.3 showing the work flow in chip planning, and the DOPs at the TE level are reflected by the particular operations within the chip planning process depicted in Fig.3.

4. Modeling Concepts

In the following, we will discuss the main concepts of the three abstraction layers of the CONCORD model. Starting with the AC level, we firstly discuss the partitioning of the overall design process into portions. Each portion reflects a clearly specified design (sub-)task that is mostly associated to a designer or a group of designers. Secondly, the cooperation primitives to coordinate the parallel work of the designers, i.e., the exchange of (preliminary) results and other design relevant data will be explained. The concepts will be illustrated by our chip planning example as described in Sect.3.

4.1 The Administration/Cooperation Level (AC Level)

As already mentioned in Sect.2, the central concept of the AC level is the design activity. It provides adequate structuring and communication primitives to model the overall design process, thus defining the basis for cooperation.

Design Activities

A ***design activity*** (DA), depicted in Fig.4a, is the operational unit realizing a design task. It can be best characterized by the following description vector consisting of four parameters: <DOT(DOV$_0$), SPEC, designer, DC>. The first parameter DOT, which stands for ***design object type***, gives the type information for the design states of that DA, i.e., for the ***design object versions*** (DOVs). All the DOVs created within a DA are organized in a ***derivation graph***, and belong to the ***scope*** of that very DA. Without further authori-

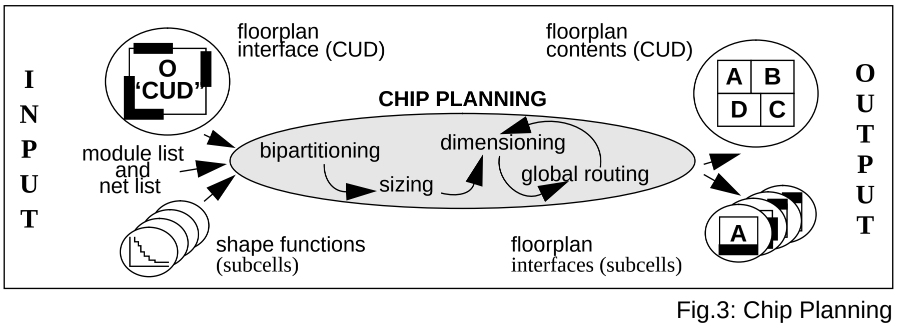


Fig.3: Chip Planning

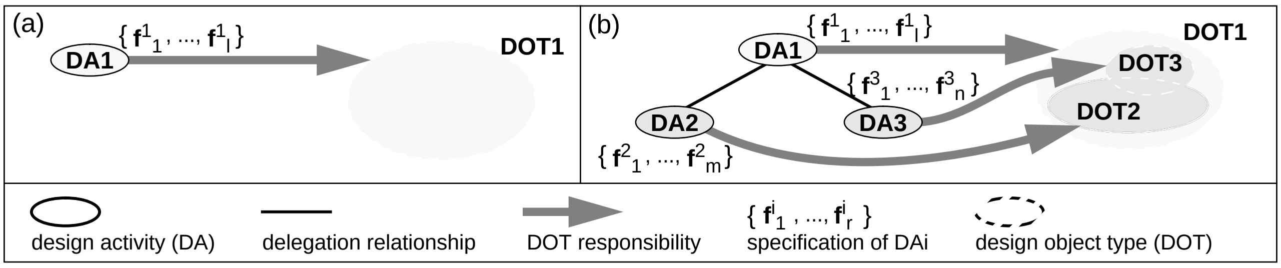


Fig.4: Design Activities and DA Hierarchies

zation a DA is only allowed to read DOVs of its own derivation graph. It is possible to initialize the scope of a newly created DA with a first DOV (DOV_0) serving as a basis for the DAs work. This is an optional add-on to the first parameter, but if specified, the DA needs to start with this first version and it will be an ancestor of all DOVs created within that DA. The design task of a DA is specified in the parameter SPEC as a set of properties the DOV to be constructed should possess. In our model, these properties are named ***features*** [Kä91]. The SPEC parameter expresses the goal of the design task and is therefore named ***design specification***. In the simplest case, a feature in the design specification of a DA constrains the value of an elementary data item to be in a certain range. A more complicated feature can express the need that the resulting DOVs have to pass a particular test tool successfully. This exemplifies that the concept of features also expresses some kind of abstraction allowing for the specification of application-specific properties relevant for design decisions. It is important to detect the *quality state* of a certain DOV, in order to ascertain the 'distance' of the current design state from the final state defined by the design specification. The quality state of a given DOV is defined by the subset of features fulfilled and is determined by the *Evaluate* operation. In the following, we distinguish *preliminary DOVs* fulfilling at most a true subset of the specification, from *final DOVs* indicating that the DA has reached its specified goal through fulfillment of the whole feature set. The third parameter assigns to each DA a *designer*. He will be responsible for the actions performed within the DA. The fourth parameter, DC, indicates that a certain design strategy has to be applied (by the designer). This topic will be detailed in Sect.4.2.

Delegation

During its efforts to reach its specified design goal, a DA may delegate parts of its own design task. This has to be done by creating sub-DAs. The execution of the *Create_Sub_DA* operation implicitly establishes a relationship called ***delegation***. It can be employed iteratively spanning a DA hierarchy as indicated in Fig.4b. There, DA1 has created two sub-DAs, DA2 and DA3. The operation *Init_Design* allows for the initiation of a design process by the creation of the top-level DA (Fig.4 a). In Fig.4, the DOT (and perhaps also an initial DOV) associated to a DA via arrow is indicated by circles (the same shade coding applies). In delegation, a sub-DA's specification always constitutes a subgoal of the super-DA's design goal. Here, the complex structure of a DOT provides a natural basis for structuring the design process. As a consequence, the DOT of the sub-DA has to be a 'part' of the super-DA's DOT. However, the specification of a sub-DA needs not to be a subset of the super-DA's specification. In general, a subgoal is not automatically derivable from the goal of the super-DA. It needs to be specified by the designer assigned to the super-DA.

Several motivations are conceivable w.r.t. the delegation of design subtasks. Among those are the following ones:

- decompose a complex design task into more practical units of work;
- transfer necessary design steps to companion designers, who are specialists for this work;
- delegate a single design task several times and choose the best of the delivered solutions;
- test alternative ways to reach the same subgoal.

The first point mentioned in that list leaves it open whether to completely split a design task or to delegate only parts of the task. In the first case, it remains as own work of the super-DA to control the design work of the subordinate DAs in the hierarchy, and to synthesize the results delivered by those sub-DAs. In the latter case, the super-DA itself has to carry out design work, and further, to integrate the results delivered by its sub-DAs with its own work. The third point mentioned in the list above indicates that the DOTs (and possibly the initial DOV) of the sub-DAs may be identical or may overlap (cf. Fig.4 b). In this case, the scopes of different DAs contain data of the same type and cooperation mechanisms (see below) have to be applied, whenever these DAs want to exchange data.

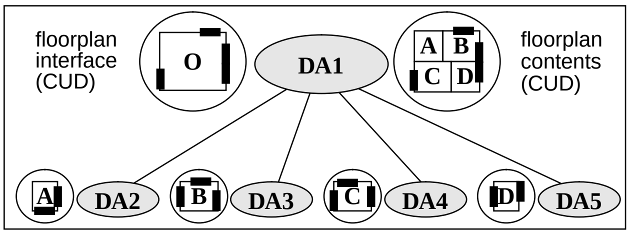


Fig.5: A Delegation Scenario within Chip Planning

The delegation concept can be illustrated by a very simplified scenario in chip planning. Fig.5 shows a sample DA (DA1) that is responsible for the planning of the initial DOV indicated by the subtree of a cell hierarchy rooted at cell O. It is further assumed that the specification of DA1 expresses features for shape/area limitations and pin restrictions for O indicated by the floorplan interface at the left-hand side of DA1 in Fig.5. DA1 starts its work by applying the chip planner tool to CUD O with subcells A, ..., D. This leads to the floorplan contents depicted at the right-hand side of DA1, which is the basis for delegating further planning steps on the subordinate hierarchy level indicated by the sub-DAs DA2, DA3, DA4, and DA5.

Cooperation Primitives

From an abstract point of view, design proceeds in a cooperative manner reflecting the conviction that a particular goal can be achieved better and in shorter time if the DAs of a DA hierarchy work together. In the CONCORD model cooperation relationships between DAs are explicitly modeled (cf. Fig.1, AC level).

The **delegation** relationship is fundamental for modeling cooperative design processes. We take up this relationship type again, because it constitutes not only a way of delegation between a super-DA and a created sub-DA, but also a way of cooperation. Remember, a DA may create an arbitrary number of sub-DAs, as long as it is appropriate to reach its own design goal. The sub-DA's termination is the precondition for the termination of the super-DA. The super-DA keeps all the rights of the creator, i.e., it is able to terminate a sub-DA (operation *Terminate_Sub_DA*) or to modify its specification (*operation Modify_Sub_DA_Specification*). Note that reformulations of design goals are typical in design applications. On the other hand, the sub-DA is only allowed to refine its own specification by addition of new features or by further restricting existing features. As soon as a sub-DA completes its work by reaching one or more final DOVs, it has to send a message to its super-DA (operation *Sub_DA_Ready_To_Commit*). The sub-DA must not terminate without the agreement of the super-DA for the following reasons. It may be possible that the super-DA wants to modify the sub-DA's specification in such a way that it would be appropriate for the sub-DA to keep the current results (design states and derivation graph) as a basis for deriving new DOVs on the way to reach the new goal. If the modification of the sub-DA's specification is not the intention of the super-DA, the sub-DA can be terminated, i.e. committed, and the final DOVs devolve to the scope of the super-DA. A further operation is *Sub_DA_Impossible_Specification*, which informs a super-DA that a sub-DA will not be able to fulfill the requirements of its specification and therefore asks for a reaction of its super-DA, e.g. termination of the sub-DA or modification of its design specification. For example, one can assume in the example of Fig.5 that after planning the subordinate levels of cell A, DA2 realizes that the specified area is not sufficient. This leads to an 'impossible specification' message from DA2 to its super-DA DA1. A possible reaction of DA1 could be to modify the specifications of DA2 and DA3 by giving DA2 more and DA3 less area. As a consequence, the planning of the cell hierarchy subtrees rooted at cell A and B will be redone using the modified area features.

Modifications of a DA specification can also be the result of negotiations between DAs. This leads to the second relevant relationship type, called ***negotiation***. The subject of this cooperation are the sub-DAs' specifications. During a negotiation process, one side may propose further refinements of the design specification and the other side may agree to or disagree with those proposals (operations *Propose*, *Agree*/*Disagree*). If two negotiating sub-DAs are not able to reach an agreement, the super-DA has to be informed (operation *Sub_DAs_Specification_Conflict*), which then has to resolve this conflict. We allow negotiation relationships between only the sub-DAs of the same super-DA, because these sub-DAs contribute to a common design goal set by their common super-DA. A detailed discussion of this cooperation model is described in [HKS92]. Negotiation relationships can be dynamically established between sub-DAs (operation *Propose*) or explicitly set by their super-DA (operation *Create_Negotiation_Relationship*). Suppose, starting from the sample scenario of Fig.5, a negotiation relationship between DA2 and DA3 is set by DA1 concerning the area for both subcells, A and B. Due to negotiation, the two connected sub-DAs are now allowed to move the borderline between A and B horizontally.

Besides the cooperation via design specification, a controlled exchange of preliminary results (design states, i.e. DOVs) of DAs is necessary. We model this data exchange by the relationship type ***usage***. A requiring DA (operation *Require*) may ask another DA (called the supporting DA) for a DOV with a certain set of features satisfied. This feature set defines the quality needed in order to express relevant design information for the requiring DA. A precondition for the usage relationship is that the requiring DA knows about the design specification of the supporting DA. From the view of the supporting DA, the delivered DOV needs not be a final one w.r.t. its own specification. A DOV becomes only visible along usage relationships, if it was propagated by its DA (operation *Propagate*). All propagated DOVs have a certain quality state determined by the operation *Evaluate*. The *Propagate* operation gives a DA control over which of its DOVs are *pre-released* and, therefore, are added to the scopes of other DAs connected via usage relationships. This means, a single DOV may belong to several scopes w.r.t. usage relationships. DAs which are not connected by a usage relationship must not exchange data.

4.2 Design Control Level (DC Level)

As briefly mentioned in the introduction, the local processing of design data within the scope of a DA is done in atomic units of work, called design operations, which can be organized by means of work-flow definitions.

Design Operations

We have already seen that design tools are mostly used to accomplish the design task associated with a DA. In order to abstract from a specific design tool, we call the action performed ***design operation*** (DOP). Thus, DAs are made up out of several DOPs that have to be executed in some specific order given by the design strategy of their DA. DOPs are used to achieve stepwise improvement of (preliminary) results, which are represented as DOVs (design states). A DOP reads several (initial) DOVs belonging to the scope of the initiating DA. It finishes by writing a resulting DOV, which does not need to be a final one. Since the specific version model and the applied notion of configurations are beyond the scope of this paper, we have taken here a simplifying, yet sufficient, view. More detailed information on DOP properties are given in Sect.4.3.

Internal Structure of a DA

Work-flow specification is the basic means to organize tool applications within a DA in order to ensure a particular design methodology. In general, it results in some pre-planning of a DA's DOP executions given by the parameter 'DC' within the description vector assigned to a DA. It defines an internal structure built upon DOPs and specific DA operations, such as the evaluation (*Evaluate*) of the quality state of DOVs, or operations serving for managing sub-DAs (e.g., *Create_Sub_DA*) and cooperation relationships (e.g., *Propose*, *Agree*/*Disagree*, *Require*, *Propagate*). This pre-planning may comprise the whole DA or only parts of it. Whenever several choices are left open or when there is a need for work-flow modifications, the associated designer (or the super-DA) has to specify how to continue using direct interventions. In the following, we describe three basic methods for work-flow specification.

One can view a design methodology as a template for

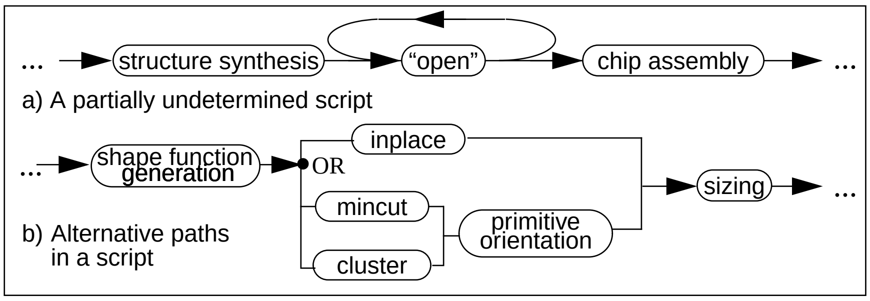


Fig.6: Sample Scripts

valid sequences of DOP executions within a DA. We call such a template a ***script***. A script usually leaves some degrees of freedom to a designer which may include choosing one of several alternative paths, performing any intermediate actions between two specified operations, perhaps containing repetitions and branches for parallel actions. Thus, a script usually allows for several concrete execution sequences. In our VLSI example (Sect.3), a DA which is to design a chip starts with the structure synthesis and ends with a chip assembly (cf. Fig.2). A script which fixes these two operations and allows for arbitrary intermediate steps is shown in Fig.6a. The use of "open" allows the specification of partially or even completely undetermined templates. In Fig.6, we employed graphical representations of scripts. Of course, a script may also be specified using a kind of programming language. A script may contain sequences, branches for concurrent execution, alternative paths as well as iterations. Fig.6b shows an example of a branch between alternative paths: after shape function generation, the designer has to decide how to proceed choosing among three alternative methods.

On the other hand, there are dependencies between the DOPs to be observed within a given design application domain (e.g., VLSI design, mechanical CAD, etc.). For instance, one may require that a DOP of a certain type (e.g., chip assembly, see Fig.2) must not be applied before a DOP of another type has successfully completed (e.g., structure synthesis, see Fig.2), or that a certain DOP must always be followed by another DOP of a specific type (e.g. pad frame editor followed by chip planner, see Fig.2). Since we define these ***constraints*** to hold for all DAs of a design application domain, any script within must not contradict these constraints.

Cooperation relationships among DAs lead to asynchronously occurring events within a DA (e.g., *Propose* or *Require* operations), generally asking the receiving DA to react or reply (e.g., *Agree/Disagree* or *Propagate* operations). For instance, one may want to define that a *Require* operation of another DA causes the current DA to look for a qualifying DOV and to (immediately) propagate, if found. Or a cooperation operation of one DA (e.g., *Modify_Sub_DA_Specification* operation) may cause another DA to stop its work at the current point and resume at another point in the script. Those kinds of specifications may be best expressed as ***(event, condition, action) rules***, since rules also correspond to exception handling in programming languages, and are best suited to cope with asynchronously occurring events. For example, the first of the above mentioned rules can be written in the following way assuming a sufficiently high-level rule language: **WHEN** *Require* **IF** (required DOV available) **THEN** *Propagate*.

Data flow

The discussion so far has concentrated on control mechanisms for work-flow specifications within a DA. However, one may also want to describe data flow between DOPs, for instance to express that a certain DOP has to continue the work of a predecessor DOP. Thus, a control flow edge between subsequent DOPs often implies data flow, too. Note however that a DOP retrieves its input from the database and stores its result in the database. Hence, the only data which needs to flow between DOPs or between DOPs and specific DA operations (e.g., Evaluate) is the identification of a DOV together with some status information[1].

4.3 Tool Execution Level (TE Level)

Although a DOP looks as an atomic operation when viewed from the DC level, it has an internal structure that provides facilities needed for long-lived (trans-) actions.

Since intermediate processing states of a DOP do not represent relevant design states w.r.t. inter-DA cooperation, it is reasonable to require that a DOP is atomic in the sense that it is performed as a whole or not at all. Furthermore, these intermediate states should not be visible outside the DOP, because only final or propagated DOVs participate in inter-DA cooperations. Whenever a DOP is finished, however, its results are to be made persistent in the form of a DOV given to the data repository. Thus, a DOP can be assigned the properties of a "classic" transaction, i.e., the ACID properties. Nevertheless, a DOP is not a "classic" transaction, because it may last for a longer span of time, for example, several hours or days in the case of a sophisticated design tool. Therefore, one has to provide an internal structure for a DOP which supports the needs of long-lasting transactions [KLMP84].

In addition to the checkin and checkout operations, there are the following structuring facilities available at the TE level (see Fig.1). *Savepoints* enable the designer[2] to "wipe out" anything he has changed later on or added to the objects under design. Consequently, intermediate states, to which a designer might wish to return later, are explicitly marked by the designer (*Save* operation). In case the designer wants to establish an earlier state, he selects the appropriate savepoint and issues a *Restore* operation. Hence, savepoints serve as a means for user-initiated rollback to reach a previously marked intermediate state. To enable a DOP to last for several days, it must be possible to suspend (*Suspend* operation) its work and resume (*Resume* operation) it after a while. The state seen by the designer after a *Resume* operation must be equal to that seen when issuing the *Suspend* command.

5. Operational Concepts and Realization Issues

After having described the most important concepts of the CONCORD model, we now want to detail on the capabilities of the activity managers that control the cooperative

1. In quite a number of cases (e.g. our chip planning DA given in Fig.3), the in-memory data structure can be handed over from one DOP to the succeeding DOP.
2. In general, the designer associated with a particular DA also supervises the DOPs to be executed in that very DA.

design process. Thereby, we review the tasks to be performed by a certain activity manager, and we discuss a manager's reactions to specific failure situations derived from an overall failure model. Before that, however, an overview of the system architecture integrating the level-specific managers will be given.

5.1 System Architecture

Design is generally performed on a network of machines, where the prevailing architecture is a workstation/server environment (connected via a local area network, LAN). Obviously, the shared design data repository and its DBMS component are located on a server, which is either embodied by a single machine or a complex of (local or distributed) machines. In contrast, the designer carries out his design work at a (single- or multi-processor) workstation.

Since a DA typically comprises the design work of a single designer, we assume that a DA is running on a single workstation. Consequently, all actions executed within a DA are managed and executed on that workstation, too. This is necessary for three reasons. Firstly, in many cases the designer has to specify input parameters for the design tools. Secondly, designer interaction during tool execution is necessary and essential, and, thirdly, the information derived by a DOP is mostly subject to work-flow (data-flow) management within the DA. Associating a DA with a workstation has direct implications to the assignment of the activity managers to their run-time (and hardware) environment. The design manager (DM) which handles the DA-internal work flow and the script-based processing is located on the workstation side. The transaction manager (TM), in turn, is responsible for the shared access of all designers and design tools to the data repository at the server, controls the data supply for workstations, and handles DOP executions. It is, therefore, split into two subcomponents. The server-TM handles checkout/checkin and controls concurrent access to DOVs, thus residing on the server, whereas the client-TM resides on the workstation managing the internal structure of DOPs. The cooperation manager (CM) has to manage the design environment set up by the cooperating DA's that are typically distributed across the workstations. However, distributed maintenance of the information incorporating the inter-DA cooperation (e.g. cooperation relationships and cooperation operations) would be overly complex due to the concurrent interactions and the distributed state information. Therefore, the CM is represented by a centralized component located at the server site, thus exploiting the global DBMS as information repository.

In case of system failures, it is important to rely on recovery concepts that keep track of the distributed design environment and its interacting system components. For that end, the hierarchically cooperating activity managers (CM, DM, as well as (server and client) TM) jointly accomplish failure handling covering all architectural levels. The TM provides *recoverable DOPs*, that is, recovery points are used for restart after a failure. The DM relies on the recoverability of DOPs and accomplishes *recoverable script executions* by relying on persistent script information. The CM, in turn, relies on the recoverability of script executions and provides *recoverability of the distributed design environment* by logging the cooperation protocols in the entire DA hierarchy.

In the following subsections, we discuss in more detail the specific tasks of each activity manager. In addition, we show the specific measures for failure handling w.r.t. a complete failure model.

5.2 The Transaction Manager (TM)

DOP Execution

As its most important task, the TM has to guarantee the ACID properties of DOPs. Due to the atomicity property, client-TM and server-TM have to accomplish a two-phase-commit protocol for all their critical interactions, i.e., for checkin and checkout, as well as for *Begin-of-DOP* and *End-of-DOP* operations. The consistency property requires that every derived DOV observes the constraints specified in the underlying database schema, and the durability (of derived DOVs) is guaranteed by the data repository, i.e. by the logging and recovery methods of the server-TM. The isolation property has to be achieved as well, but needs some more clarifications. Concurrent work of multiple DOPs on the same DOV is conceivable for the following two cases:

- The DOPs were initiated by a *single* DA with the shared DOV belonging to that DA's scope. In this case, each DOP is expected to derive a new version concurrently and to modify the DAs (single) derivation graph. This modification is done within a DOP's checkin operation and therefore the TM has to protect the proliferation of the DA's derivation graph, e.g. employing a locking protocol based on short locks.
- The DOPs were initiated by *multiple* DAs with the shared DOV derived in one DA and with the other DAs being authorized to read this DOV due to established usage relationships. In this case, the DOPs that concurrently work on that DOV are to derive separate new versions that make it to their own DAs' derivation graphs, which are disjoint from each other, thus preventing write conflicts.

In addition to that, a DA may acquire a *derivation lock* on a certain DOV to prevent multiple checkout (and concurrent processing) of this DOV for application-specific reasons. In this case, the (server-) TM has to provide long-lasting isolation for that DOV (e.g., long locks usually managed at the server site). In contrast, short locks are fully sufficient to protect a checkin or checkout operation. Obviously, we heavily used the underlying versioning and version derivation concept in order to achieve information (DOV) sharing as well as isolated execution (DOV derivation).

Failure Handling

A *system failure* is typically caused by a crash of workstation or server. The impact of a server crash can be minimized to only affect the server site due to a workstation-server interface that is carefully designed according to maximum isolation between server and client components (TM) as discussed in more detail in [HHMM88]. On the other side, a workstation crash affects all the activities currently running on that very machine. As a consequence, it causes the loss of the context[1] associated to an active DOP. Since DOPs are long-lived transactions, it is inadequate to treat system failures by rollback to the very beginning. Instead, they need a stable processing environment. Hence, system failures are handled by partial rollback to *recovery points*. Recovery points act as "fire-walls" inside a DOP that limit the scope of work lost in case of a failure and provide a starting point after recovery [HR87]. These recovery points are

1. The context of a DOP consists of the current state of the design data and on information about the state of the application program implementing the DOP.

chosen automatically by the system after appropriate events or time intervals and are transparent to design tool and designer. In particular, after each checkout operation a recovery point is set in order to avoid duplicate requests of a DOV from the server in the case of a failure. Furthermore, the mechanisms of recovery points are used to implement the savepoint concept (see Sect.Sect.4.3). A recovery point makes the current DOP context persistent. In order to cope with system failures, the TM has to rely on the most recent recovery point. A more detailed discussion of handling the internal structure of DOPs is given in [HHMM88].

Commit and Abort

There are two operations initiating or finishing the DOP processing, i.e., the operations *Begin-of-DOP* and *End-of-DOP*. The first one is given by the DM (i.e., design manager at the superordinate design control level) to indicate the start of a new DOP (to the client-TM). Usually, it is accompanied by the start parameters. Since the latter operation has to cover two different outcomes of the DOP execution, it is split into two separate operations. Whenever a DOP encounters an inconsistent state or is not successful for some reasons (sometimes even determined by the designer), it will abort its activities. On the other side, if the DOP reaches a final state, it issues a commit operation to close its processing. For both, the *commit* and *the abort* operation, the server-TM is firstly asked to release the derivation locks held (if any), then the client-TM removes all its savepoints and its recovery point, and finally gives the appropriate message (sometimes accompanied by some return parameters) to its DM.

Checkout and Checkin

Checkout and Checkin operations are separated from *Begin-of-DOP* and *End-of-DOP* operations, respectively. A checkout operation is assumed to read a DOV from the server DBS. At this point it has to be tested that, firstly, the DOV belongs to the scope of the DOP's DA, and, secondly, there is no incompatible derivation lock on the DOV. In case of a successful checkout, the appropriate derivation lock is set to achieve proper protection. The checkin operation behaves as the complement to the checkout operation. It gives the derived DOV to the DBMS for storing. The consistency of the newly created DOV has to be checked and further, its DA's derivation graph is extended by the newly created DOV, since this DOV now belongs to the scope of that DA.

In addition to these cases, a TM has to be able to deal with the situation that the checkin operation isn't successful due to problems at the server site. This situation will occur, for example, if a DOV was created that doesn't fulfill the integrity constraints which are to be enforced by the server DBMS. In this case, the server-TM has to inform the client-TM which, in turn, has to indicate this 'checkin failure' situation to the DM, or some automatic (programmed) actions take place, e.g. subsequent abort or commit[1].

5.3 The Design Manager (DM)

The DM has to enforce the work flow within its DA and to handle external events caused by cooperating DAs. From a system point of view, work-flow execution may be done similar to ConTracts [WR92]. The most important point here is to be able to restore the most recent consistent processing context in the case of a system failure as basic means to continue processing with a minimum loss of work. In the following, we will discuss these topics in more detail.

Work Flow Management

As already mentioned in Sect.4.2, a DA's work results from its script, the active rules, and the given constraints. Whenever the work flow is unambiguous, the DM provides automatic execution. This means that the DOPs are started as defined by the work flow, provided all DOP parameters are available (esp. the identifier of the input-DOVs). As soon as a DOP finishes, the TM passes on the information needed by the DM to proceed, i.e., commit/abort flag and a handle to the DOP's design data (e.g., the identifier of the output-DOV).

In general, however, a fully automatic processing is not possible. Work flow often depends on creative design decisions which are to be taken during the design work and cannot be preplanned. For that reason, incomplete work-flow specifications might become sufficient. In these cases, a continuation of processing mostly requires designer interaction. A designer might be prompted to either provide missing parameters for DOP executions, or to chose out of several, alternative continuation possibilities. Beyond that, the designer is allowed to step in (at certain points of script execution) and cause the iteration of a sequence of executed DOPs (starting from a different DOV or using modified input parameters for DOP executions).

Coping with External Events

A change in the current work flow processing might be necessary due to some external events. The first class of events refers to the delegation relationship set between DA's (see Sect.4.1). Whenever the DA's description vector is modified by the super-DA (operation *Modify_Sub_DA_Specification*) or the current specification is impossible to fulfill (operation *Sub_DA_Impossible_Specification*), DA execution has to be restarted from the beginning. However, the designer may choose any previously derived DOV as a starting point for the new activation. Another important class of external events is the withdrawal of a pre-released DOV by a supporting DA (see Sect.4.1, usage relationships). The DM of the requiring DA has to analyze (its log data, see below), whether the pre-released DOV was used within a local DOP thus affecting locally derived DOVs. If this is the case, the processing needs to be stopped and the designer has to decide on how to continue. Designer interaction has already been discussed above. Note, there is no necessity for the designer to invalidate his own results, if he concludes, for example, that his current work is not negatively influenced by that withdrawal.

Failure Handling

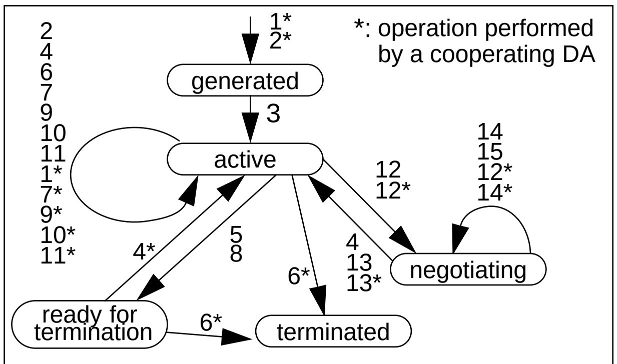
The DM is responsible for failure handling within a DA. The system failure which affects the DM is the crash of the workstation it runs on. After restart of the workstation, the DM has to recover the last consistent state of DA execution, in order to continue the script processing. Here, the DM relies on the recoverability of the DOPs that has also been affected by the workstation crash (see Sect.5.2). This requires, in addition to a persistent script, the DM to log system activities. A log entry capturing all DOP parameters is written for each start and finish of a DOP execution. All interactions between DM and TM are, again, accomplished by means of safe communication[2]. By means of persistent script and persistent log the DM is able to provide a forward-oriented con-

1. In treating this situation as a commit, i.e., as a valid DOP termination, we allow a kind of data flow between subsequent DOPs because the current design state is still available. As a result, the data flow between DOPs is not restricted to completely derived DOVs. This issue will not be detailed in this paper.

text management in case of system failures.

5.4 The Cooperation Manager (CM)

The CM embodies the mediator between cooperating DAs. It enforces that cooperation takes place only along established cooperation relationships, and it further checks each cooperative activity to comply with the integrity constraints of the underlying cooperation relationship (discussed in Sect.4.1). In order to do this, state information about each DA in the hierarchy has to be maintained. This information includes the description vector and the scope of each DA as well as the established cooperation relationships. In the following, we will focus on two important issues: state transitions of DAs and cooperation correctness due to visibility of preliminary information.



1	Init_Design	9	Propagate
2	Create_Sub_DA	10	Require
3	Start	11	Create_Negotiation_Rel
4	Modify_Sub_DA_Spec	12	Propose
5	Sub_DA_Ready_To_Commit	13	Agree
6	Terminate_Sub_DA	14	Disagree
7	Evaluate	15	Sub_DA_Spec_Conflict
8	Sub_DA_Impossible_Spec		

Fig 7: Simplified State/Transition Graph for a DA

Cooperation Control by Means of State Transitions

In order to enforce proper DA reactions, different states are distinguished within the lifetime of a DA (cf. Fig.7). The state *generated* is assigned to a DA when it already got initiated via a description vector, but hasn't begun its work so far. In the *active* state a DA performs its design work. The state *negotiating* is assigned to a DA whenever it is requested to negotiate or wants to negotiate itself. As soon as a DA changes to the state *negotiating*, its internal processing is suspended, and after returning to the *active* state, internal processing is resumed, maybe with a modified design specification. Of course, the associated designer can take over control and properly react to the modifications agreed upon (see Sect.5.3). Note, continuing the design work during negotiation might waste time and efforts.

After a DA has created a final DOV it should not be terminated until the super-DA has accepted its result. Further, it should not do any more work until the super-DA has issued a corresponding request. We cover this situation with the state *ready for termination*. This state will also be assigned to a DA which notified its super-DA that it will not be possible to derive a final DOV satisfying the current design specification. The state *terminated* indicates that a DA has been terminated by its super-DA and vanished from the DA hierarchy. The state transition graph shown in Fig.7 illustrates the different states of a DA. There, events issued by other (cooperating) DAs are marked by an asterisk.

2. This may be achieved by transactional RPC or by a specialized two-phase-commit protocol [GR93].

Controlling the Dissemination of Preliminary Design Information

The CM enforces limited visibility of DOVs by means of the relationship types delegation and usage for which the following restriction holds: a DA is only allowed to see DOVs, which belong to its scope[1]. Otherwise, a committed DOV would become generally accessible. Therefore, we looked for a salient approach to control that kind of dissemination. At the first sight, it seemed that the required protection could be achieved by some access control mechanisms. However, such mechanisms typically do not support inheritance. Furthermore, they do not seem suitable because of the high dynamics and the request flexibility needed in the system. Therefore, we decided to develop a locking scheme which uses an inheritance mechanism similar to that used in nested transactions [Mo81]. It is important to distinguish passing on DOVs within a DA from among DAs. In the first case, a DOP has created a new DOV via checkin operation. As mentioned in Sect.5.2, this newly created DOV becomes part of its DA's scope simply by inserting it into the DA's derivation graph. It is guaranteed by means of *scope-locks* that a DA's derivation graph is isolated. Hence, the DOVs derived in a DOP are passed on to only that DA. The second case deals with restricted transfer of DOVs among DAs. Transfer along delegation relationships is enabled via inheritance of scope-locks, whereas transfer along usage relationships is managed by scope-lock compatibilities. Referring to delegation relationships a super-DA inherits the scope-locks on the *final* DOVs of its terminated sub-DAs and then retains these locks. After finishing the top-level DA all locks are released. The inherent differences to a locking scheme for nested transactions are implied by the processing and cooperation constraints of the CONCORD model:

- difference in inheritance: only locks on final DOVs are inherited, and a super-DA may read the final DOVs of a sub-DA as soon as the sub-DA changes its state to ready-for-termination;
- difference on isolation: a lock may be granted to a DA if it has a usage relationship to the DA which retains the lock (provided that the particular DOV has been propagated and fulfills the required quality state).

Invalidation and Withdrawal of Pre-Released Design Information

Whenever some preliminary design information got pre-released, but later on changed or even removed, the CONCORD system has to react properly in order to guarantee a minimum of consistency. There are two cases to be distinguished. *Invalidation* of pre-released design information, i.e., of propagated DOVs along a usage relationship, is given as soon as it becomes clear that a pre-released DOV will not be an ancestor of a final DOV in the supporting DA. In this case, another DOV from the scope of that DA which fulfills all the required (and possibly more) features of the previously propagated DOV will be propagated by the CM to the requiring DA for replacement. The other situation is characterized by a *withdrawal* of pre-released design information. If the DA is cancelled or the specification of the DA

1. Recall, a DA's scope has been defined to include the DOVs of its derivation graph, the final DOVs of its terminated sub-DAs, and the DOVs that became visible along its usage relationships (see Sect.4.1).

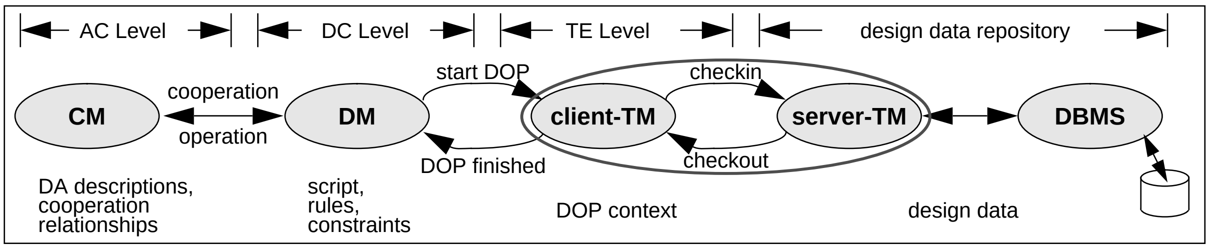


Fig 8: Responsibilities and Interplay of Activity Managers

is changed such that the features of a previously propagated DOV are not part of a new specification, the propagation has to be withdrawn. This causes the CM to send a notification to all the (requiring) DAs that have seen that DOV subject to withdrawal. After having received this 'withdrawal' notification, the DAs (and their DMs or associated designers) can react to this notification by means of the mechanisms already discussed in Sect.5.3.

Failure Handling

The CM is also responsible for failure handling related to the inter-DA structure. Here, we have to consider the system failures crash of workstation and crash of server. To react to a server crash, the CM only needs to hold persistent the DA-hierarchy-describing information mentioned at the beginning of this section. To that end it can employ the data management facilities of the server DBMS. For all communications between the nodes of the LAN, we assume reliable communication protocols (transactional RPC, see Sect.5.3) which insulate the cooperation protocols from network failures and workstation crashes. Hence, a workstation crash does not affect the CM.

5.5 Interplay of Activity Managers

In this section we tried to clarify the tasks as well as the interplay of all activity managers involved at the different layers of the CONCORD model. As a kind of summary, Fig.8 emphasizes our approach to joint activity management and joint failure handling spanning all architectural levels. Activity management including failure handling relies on a sophisticated reliability concept that consists of communication and execution/control reliability. The first one refers to interactions between activity managers, where reliable communication protocols are accomplished by means of transactional RPC or by a specialized two-phase-commit protocol [GR93]. The latter one refers to the reliability of each manager's control sphere.

6. Conclusions

In this paper, we described the CONCORD model, which is our approach towards support for cooperative design. The CONCORD model provides means for performing tool-based design organized by certain design methodologies and allows to reflect the major characteristics of design processes such as goal orientation, hierarchical refinement, stepwise improvement as well as team-orientation, and cooperation.

As outlined in the course of the paper, the CONCORD model was derived by analyzing cooperative design methodologies. As a result from that, we followed a top-down approach starting with a cooperation layer that is built upon an activity model which, in turn, embeds classical ACID transactions. We designed the CONCORD model along those lines to be a layered approach that provides layer-specific modeling and controlling concepts that are used by the superordinate layer as primitives to build upon:

- The AC level embeds cooperation semantics and supports application-specific exchange of (preliminary) design information under system control.
- The DC level allows the application of certain design strategies and provides mechanisms for specification and control of work flow.
- The TE level encapsulates long-living, yet atomic, units of design work.

It is the fruitful combination of these concepts that gives the model its flexibility and power. This top-down approach may be the reason for the discrepancy between our results and the capabilities of so-called extended transaction models which are typically developed in a bottom-up manner. In contrast to those other approaches, we built upon practically approved transaction concepts (ACID transactions, ConTracts capabilities) and tried to separate the concepts into the three consecutive layers according to their abstraction capabilities. Consequently, we can take advantage of inherent properties of a layered system approach, i.e., complexity reduction, separation of responsibilities, isolation of usages and testing, adaptability etc.

In a general setting, we can claim that there is a real need to have all three levels. However, in specific cases it might be sufficient to use only the capabilities provided at the TE level or at the DC level, but not, for example, the concepts available at the AC level. Those specific usages are directly supported by the CONCORD approach due to its layered architecture.

Throughout the paper, we tried to provide a fairly detailed discussion of the salient concepts of the CONCORD model. However, there is still more work to do especially concerning an efficient realization. We mentioned the concept of transactional RPC as a means to provide a save communication protocol. However, it is clear to us that its realization should exploit the most efficient concepts available in the system (hardware) environment currently at hand. For example, we can use the (X/OPEN) two-phase-commit protocol and its optimization alternatives [SBCM93] for LAN communications (e.g. CM-DM communications and (client-TM)-(server-TM) communications). In case of local communications within the same machine (e.g. DM-TM communications) we can use the same mechanism but implemented more efficiently based on main memory communication.

Although the CONCORD model is not yet fully operational, we have already gained some first practical experiences. The state of realization can be described as follows:

- Parts of the AC level have been implemented in such a way that all the level-specific context data is managed by the design data repository. Initial 'in-the-field' experiments validating the modeling concepts of the AC level

have been run in the design areas of VLSI and software engineering. The results are very promising and will be published in separate papers. Still missing is cooperation control.
- Concerning the DC level, we are pretty confident that we can benefit from the implementation and application experiences of the work done on ConTracts [WR92]. Therefore, we do not put high priority on that area.
- A first implementation of the TE level [HHMM88] is operational on our design data repository (exhibiting a flexible version concept [KS92] realized on the non-standard DBMS PRIMA [HMMS87]).

In the future, we will concentrate on the AC level as well as on inter-level coordination. The latter aspect involves not only the collaboration of the level-specific activity managers, but also efficiency and optimization considerations among the managers participating in joint work and failure handling. Another aspect to be considered is the natural heterogeneity of the design environment. For simplicity purposes, in this paper we assumed one logical server, i.e. a central data repository under control of a DBMS component. Especially w.r.t. pre-existing tools this view has to be refined. A realistic approach needs to consider distributed data management by heterogeneous facilities in order to support data exchange and interoperability of these tools. Since CONCORD has been designed to be a distributed, transactional system we assume that heterogeneous and distributed data management does not influence the major model of operation. Certainly, TM as well as CM have to be adapted to these characteristics. This will be another focus of future work.

Acknowledgments

We would like to thank Wolfgang Käfer for his comments on an earlier version of this paper and Joachim Thomas for his enthusiasm in finding the CONCORD acronym.

7. Literature

El92 Elmagarmid, A. (ed.): Transaction Models for Advanced Database Applications, Morgan Kaufmann, San Mateo, CA, 1992.

GR93 Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques, Morgan Kaufmann Publ., San Mateo, CA, 1993.

GS87a Greif, I.; Sarin, S.: Data Sharing in Group Work, in: ACM Transactions on Office Information Systems, Vol. 5, No. 2, pp. 187-211.

GS87b Garcia-Molina, H.; Salem, K: Sagas, ACM SIGMOD, 1987, pp. 249-259.

Hä89 Härder, T.: Non-Standard DBMS for Support of Emerging Applications - Requirement Analysis and Architectural Concepts, (Eds: Shriver, B.D.), Proc. of the 22nd Hawaii Int. Conf. on System Sciences (HICSS-22), Kailua-Kona, Hawaii, Volume II, 1989, pp. 549-558.

HHMM88 Härder, T., Hübel, C., Meyer-Wegener, K., Mitschang, B.: Processing and Transaction Concepts for Cooperation of Engineering Workstations and a Database Server, in: Data and Knowledge Engineering 3, 1988, pp. 87-107.

HHZB92 Heiler, S.; Haradhvala, S; Zdonik, S.; Blaustein, B.; Rosenthal, A.: A Flexible Framework for Transaction Management in Engineering Environments, in: Transaction Models for Advanced Database Applications, Morgan Kaufmann, San Mateo, CA, 1992, pp. 87-122.

HKS92 Hübel, C., Käfer, W., Sutter, B.: Controlling Cooperation Through Design-Object Specification - a Database-oriented Approach, in: Proc. of the European Design Automation Conference, Brussels, Belgium, 1992.

HR83 Härder, T., Reuter, A.: Principles of Transaction-Oriented Database Recovery, ACM Computing Surveys 15, 4, 1983, pp. 287-318.

HR87 Härder, T., Rothermel, K.: Concepts for Transaction Recovery in Nested Transactions, in: Proc. ACM SIGMOD'87 Conf., San Francisco, 1987, pp. 239-248.

KLMP84 Kim, W., Lorie, R., McNabb, D., Plouffe, W.: Nested Transactions for Engineering Design Databases, in: Proc. 10th VLDB Conf., Singapore, 1984, pp. 355-362.

KPE92 Kühn, E.; Puntigam, F.; Elmargarmid, A.K.: Multidatabase Transactions and Query Processing in Logic, in: Transaction Models for Advanced Database Applications, Morgan Kaufmann, San Mateo, CA, 1992, pp. 297-348.

Kä91 Käfer, W.: A Framework for Version-based Cooperation Control, Proc. of the 2nd Symposium on Database Systems for Advanced Applications (DASFAA), Tokyo, Japan, 1991, pp. 527-535.

KS92 Käfer, W., Schöning, H.: Mapping a Version Model to a Complex Object Data Model, Proc. of the 8th Int. Conf. on Data Engineering, Tempe, Arizona, 1992.

Mo81 Moss, J.E.B.: Nested Transactions: An Approach To Reliable Computing, M.I.T. Report MIT-LCS-TR-260, M.I.T., Laboratory of Computer Science, 1981.

NRZ92 Nodine, M.H.; Ramaswamy, S.; Zdonik, S.B.: A Cooperative Transaction Model for Design Databases, in: Transaction Models for Advanced Database Applications, Morgan Kaufmann, San Mateo, CA, 1992, pp. 53-86.

PKH88 Pu, C.; Kaiser G.; Hutchinson, N.: Split-Transactions for open-ended Activities, in: Proc. of the 14th VLDB Conf., Los Angeles, 1988, pp. 26-37.

RMB92 Rodden, T.; Mariani, J.A.; Blair, G.: Supporting Cooperative Applications, Int. Journal on Computer Supported Cooperative Work, Kluwer Academic Publishers, Vol. 1, Nos. 1-2, 1992, pp 41-69.

RSJK93 Reddy, Y.V.R.; Srinivas, K.; Jagannathan, V.; Karinthi, R.: Computer Support for Concurrent Engineering, in: IEEE Computer, January 1993, Vol. 26, No. 1, pp. 12-16.

SB92 Schmidt, K.; Bannon, L.: Taking CSCW Seriously, Int. Journal on Computer Supported Cooperative Work, Kluwer Academic Publishers, Vol. 1, Nos. 1-2, 1992, pp. 7-39.

SBCM93 Samaras, G.; Britton, K.; Citron, A.; Mohan, C.: Two-Phase Commit Optimizations and Tradeoffs in the Commercial Environment, in: Proc. of the 9th Int. Conf. on Data Engineering, Vienna, Austria, 1993, pp. 520-529.

Sk91 Skarra, A.: Localized Correctness Specifications for Cooperating Transactions in an Object-Oriented Database, Office Knowledge Engineering, Vol. 4, No. 1, 1991, pp. 79-106.

US92 Unland, R.; Schlageter, G.: A Transaction Manager Development Facility for Non Standard Database Systems, in: Transaction Models for Advanced Database Applications, Morgan Kaufmann, San Mateo, CA, 1992, pp. 399-465.

WR92 Wächter, H.; Reuter, A.: The ConTract Model, in: Transaction Models for Advanced Database Applications, Morgan Kaufmann, San Mateo, CA, 1992, pp. 219-264.

WS92 Weikum, G.; Schek, H.J.: Concepts and Applications of Multilevel Transactions and Open Nested Transactions, in: Transaction Models for Advanced Database Applications, Morgan Kaufmann, San Mateo, CA, 1992, pp. 515-554.

Zi86 Zimmermann, G.: PLAYOUT - A Hierarchical Layout System, in Proc. of the 18th GI Conference, Hamburg, Springer-Verlag, Informatik-Fachberichte 188, 1988, pp. 31-51.

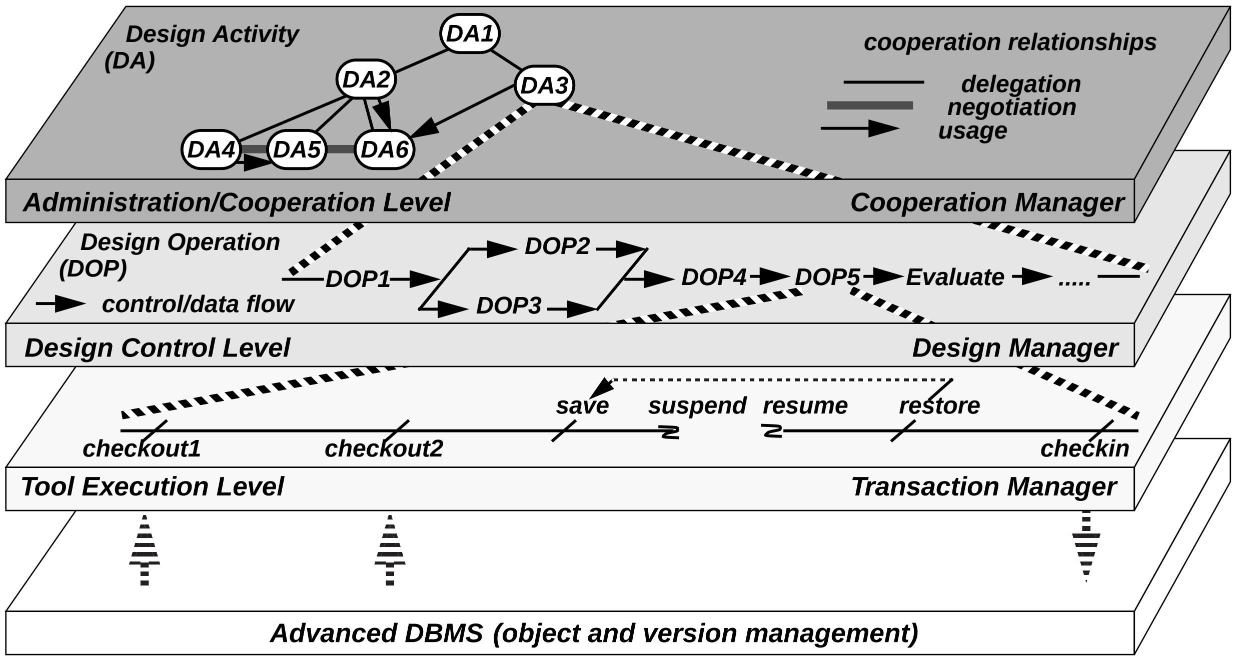
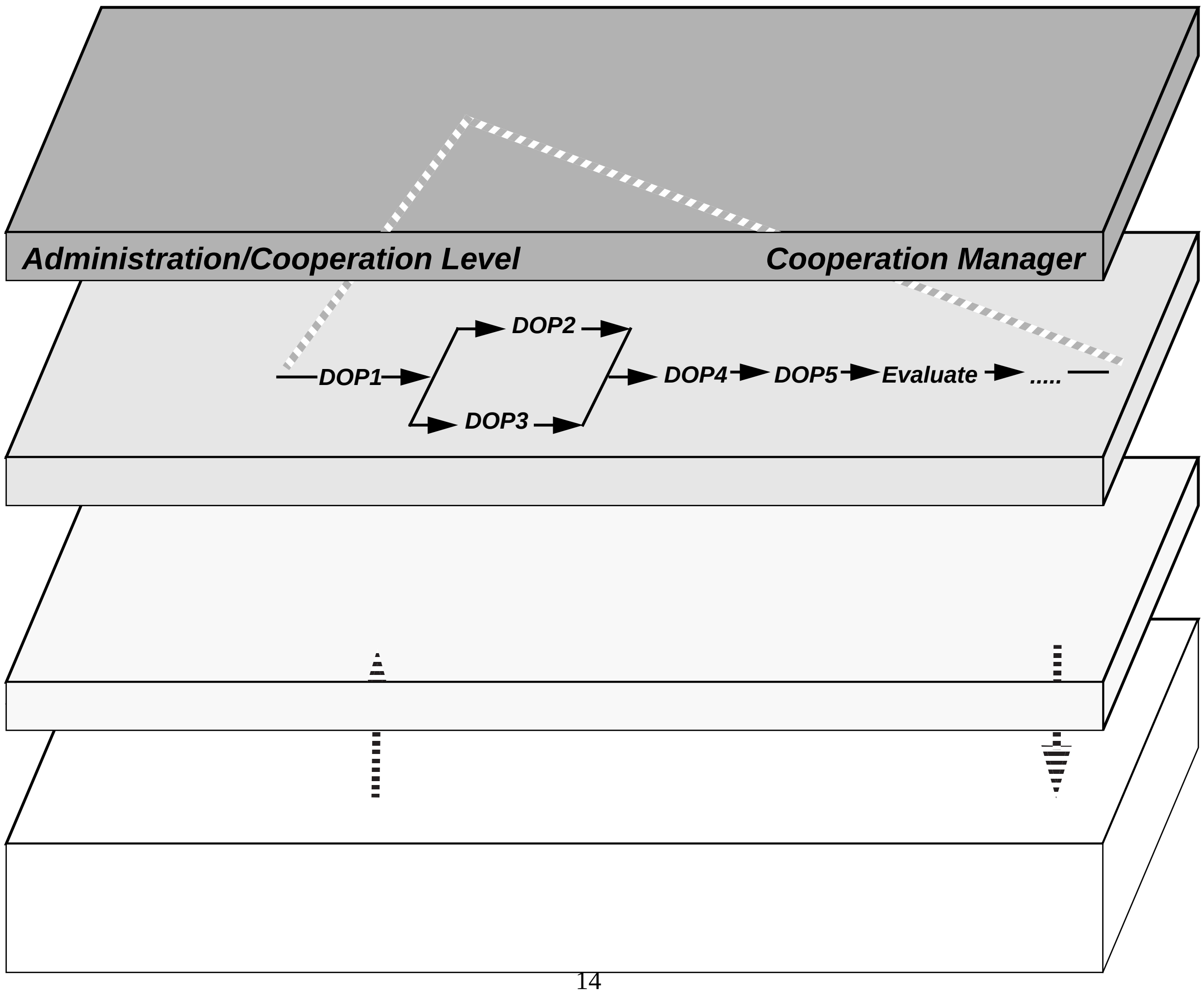


Fig. 1: Abstraction Levels of the CONCORD Model



Administration/Cooperation Level
Cooperation Manager
DOP1
DOP2
DOP3
DOP4
DOP5
Evaluate
.....