Conflict Management in CONCORD

Norbert Ritter*, Bernhard Mitschang**, Theo Härder*

** Department of Computer Science*
University of Kaiserslautern
P.O.Box 3049, 67653 Kaiserslautern, Germany
e-mail: ritter|haerder@informatik.uni-kl.de

*** Department of Computer Science*
Technical University Munich
Arcisstr. 21, 80290 Munich, Germany
e-mail: mitsch@informatik.tu-muenchen.de

Abstract

Amongst the most important components in concurrent design/engineering environments is a flexible and sophisticated activity management. Due to the fact that designers and design groups generally work in parallel on a common design task, thus performing collaborate work, activity management on one hand has to support concurrent design, whilst on the other hand it has to deal with any kinds of design conflicts either on basic design data, derived design objects, or even on applied design strategies or design goals. Since such undesired events and conflicts may occur, an adequate conflict management has to be provided. In this paper, we discuss the concepts and principles of conflict handling in design environments. As a result of our considerations, we propose a layered conflict management architecture to be integrated with the existing activity management of a CAD framework. As an example, we elaborate this architecture in the context of our CONCORD system.

Keywords: Activity/Conflict Management, Cooperation, Concurrent Engineering, Design-Flow, Transactions.

1. Introduction

1.1 Motivation

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 co-operating with each other, e.g., by negotiating their individual design goals or by exchanging their partial results. Consequently, several approaches have been proposed in the literature to enhance CAD frameworks [Ha90] by adequate facilities for activity support comprising for example cooperation-control facilities [SSZ94, MHR96], design-flow management facilities [Bo95, Br93b] or design-transaction processing [El92].

Generally, an activity model must fulfill the following requirements. Firstly, it must provide an adequate structuring of processing, i. e., the processing units (e. g. transactions, workflows, design tasks etc.) must reflect the semantics of corresponding real-world processes and, thereby, must allow an adequate mapping of real-world processes to system processes. Secondly, the activity model must foresee possible conflicts which may occur within or between the processing structures provided. Hence, a conflict model and a corresponding conflict-management are among the vital constituents of each activity model. Conflicts may be roughly classified into internal conflicts and semantic conflicts. Examples for internal conflicts taken from the realm of database systems are all multi-user anomalies (e. g. lost update [GR93]) which are prevented by well-known concurrency-control protocols [BK91]. We have named these kinds of conflicts internal since they are locally detected and treated (resolved) in an automatic fashion by a local component. Semantic conflicts are application-specific. For example, a design-flow management system allows to specify contingency design-flows which are carried out by the system, if a certain (pre-specified) design quality has not been reached at a certain design state. Here, new techniques of conflict management must be provided, allowing the application to specify semantic conflict situations as well as corresponding reactions.

1.2 Providing Conflict Management for Design Environments

Due to the fact that designers and design groups generally work in parallel on a common design task, activity management on one hand has to support concurrent design, whilst on the other hand it has to avoid any kinds of design conflicts either on basic design data, derived design objects, or even on applied design strategies or pursued design goals. Generally speaking we can view a (semantic) **conflict** as a situation within a design, where some constraints on the design process itself or on associated design data are not met. This can be the case in a number of different situations as for example the following scenarios:

- cooperation conflicts: actions on cooperation data are in contradiction with the goal of a cooperating task;
- design-strategy conflicts: the chosen strategy to solve a design task does not lead to expected intermediate states or to the requested result;
- data-access conflicts: incorrect modifications on shared design data.

Regarding such conflict situations, the following questions must be answered. Which situations raise conflicts? Which conflicts can be avoided and how can avoidance be achieved? How can those conflicts, which cannot be avoided, be detected and which reactions are necessary to resolve those conflicts? How can users specify application-specific conflict situations and corresponding reactions?

1.3 Objectives and Overview of the Paper

In this paper, we want to answer the questions mentioned above. We report on a flexible conflict-management model that can be easily integrated with a sophisticated activity management as being used in a design environment or even in a CAD framework. The objectives of the paper are:

- analyzing conflict-management concepts and proposing feasible concepts for (complex) design models;
- proving the importance of exploiting conflict avoidance/prevention;
- proving the importance of exploiting flexible reactions, if conflicts cannot be avoided/prevented;
- discussing the approach taken in the CONCORD processing model [Ri94] as an example for the realization of an adequate conflict management as an integral part of a sophisticated activity management for design processes.

In order to reach these objectives the paper is organized as follows. The basic definitions that clarify the notion of ‘conflict’, ‘conflict model’, and ‘conflict management’ as well as enlightening examples and comparisons to related work will be given in Sect. 2. In Sect. 3, we give a short overview of the CONCORD system and subsequent Sect. 4 introduces conflict management as exploited within the CONCORD cooperative processing model. There we detail on local conflict management at the various CONCORD system layers. Sect. 5 reports on integrated conflict management comprising the various CONCORD layers. Finally, Sect. 6 gives a conclusion and an outlook to further work.

2. Notion of Conflict Model and Conflict Management

As stated above, conflict management is among the key issues of activity management, because it determines processing correctness as well as data consistency. Conflict management depends on the ‘conflict model' describing possible conflicts and the corresponding system behavior. In the following, we give a very general and broadly applicable description of a conflict model, from which the definitions of conflict as well as (for the task) of conflict management can be derived. We clarify and detail our understanding of these concepts by focussing on the constituents that make up the conflict model.

2.1 Basic Definitions

To introduce the basic notion of 'conflict' we, first of all, consider a self-contained system. Later in this section we will extend the basic definition to a layered system, which can be regarded as consisting of a number of subsystems.

*Definition '**System**':*

The (self-contained) *system* S is defined as S = (O, C, M, A), where O is a set of *objects*, M a set of object-manipulating *operators*, A a set of *processing-unit types*. Each processing-unit type is given as a sequence of operators contained in M and represents a certain kind of real-world process. C is a set of *constraints*, the system has to assert. •

The notion of *constraint* is the major constituent of conflict management. The set C of constraints can be divided into data constraints and processing constraints. A *data constraint* dc can be described as dc = (IC, O_{dc}, A_{dc}) with IC being a condition, O_{dc} (the objects on which the condition must be asserted) being a subset of O and A_{dc} being either a subset of A or a subset of M referring to the actions which must not invalidate dc. Obviously, this definition is more general than what is commonly known as integrity constraints in DBMS, since the data scope can be assigned explicitly. For example, an integrity constraint guaranteeing that the salaries of all managers are always the same can be expressed as

dc = (IC={salary(mgr_1) always equals salary(mgr_2)}, O_{dc}={all managers}, A_{dc}={UPDATE(salary)}).

Any data model inherent constraints as e.g. primary key and foreign key constraints can be expressed similarly.

Processing constraints, on the other hand, define allowed orderings of actions. Actions can be processing units or operations issued within processing units. Thus a processing constraint pc can be defined as pc = (A_{pc}, R_{pc}) with A_{pc} being either a subset of A or a subset of M and R_{pc} defining a partial order over A_{pc} which must not be invalidated during run time. For example, again in the realm of DBMS, typical processing constraints are given by a concurrency control protocol that describes which (read/write) operations are allowed subsequent to earlier (read/write) operations.

*Definition '**System State**':*

A *system-state* S_i is defined as S_i = (S, H). Besides the underlying system S the definition contains a history H of either processing units or operations. Processing units are instantiations of processing-unit types, operations are instantiations of operators. H led the system from the initial state to the current state. We define a *legal system-state* as a state which fulfills the following two conditions:

(1) *Correctness-Condition*: Each processing unit within H encapsulates an *allowed* sequence of operations and H itself is given as an *allowed* set of (partially) ordered actions. 'Allowed' means not violating any processing constraint.

(2) *Consistency-Condition*: None of the processing units contained in H as well as none of the operations executed within a processing unit did violate any data constraint. •

Exploiting the definitions given so far, we can now define what a ‘conflict’ is.

*Definition ‘**Conflict**’:*

A conflict is given whenever an operation or a processing unit directly or indirectly violating a constraint contained in C has been executed or shall be executed. Directly means that the effect of the single action violates or would violate a constraint in C; indirectly means that the action supplements the history in a way that it contains a sequence of actions violating a constraint. •

This definition introduces the notion of conflict in a very broad sense. It captures conflicts,

- which are caused by violations of general rules; for example, a data model inherent rule demands not to violate referential integrity or an activity model inherent rule enforces not to read an object, which has been updated by an uncommitted transaction;
- which are caused by violations of application-specific rules; for example, a data-related rule enforces not to violate a particular semantic integrity constraint or an activity-related rule requires not to start a design tool before a second tool has been terminated successfully.

The simple examples given above show that our definition of ‘conflict’ still holds when regarding generally known data models and activity models, but as we will see in the following it is flexible enough to provide a universal view to conflict management.

*Definition ‘**Conflict Management**’:*

The term conflict management captures mechanisms suitable to handle conflicts. Thus, we define conflict management as being either *conflict avoidance* or *conflict detection with subsequent conflict resolution*. •

Conflict avoidance means that model-inherent mechanisms or application-specific mechanisms prevent certain conflicts to occur. For example, if the data model provides (explicit) versioning, less conflicts occur during concurrent writing, because different processing units may write different versions of a shared object. We call this (data-model inherent or activity-model inherent) kind of conflict avoidance *general avoidance*. Besides general avoidance, *application-specific avoidance* may be applicable. This kind of avoidance can be achieved by application-specific a-priori specifications. For example, if the application pre-plans the schedule of certain processing units and there is a scheduler enforcing these specifications, processing conflicts cannot occur (since specifications are assumed to be correct).

Often conflicts cannot be avoided. Then, the only alternative is to detect and resolve them in order not to risk specific inconsistencies. *Conflict detection* can further be divided into *foresighted detection* and *subsequent detection* (w.r.t. actions). *Foresighted detection* means that the intension of executing a certain action must be submitted and it can be decided in advance (i.e. before execution) whether or not a constraint will get violated. *Subsequent detection*, on the other hand, means that the action is executed and the system state reached must be checked to be consistent. Subsequent detection requires a resolution consisting of two steps; firstly, transferring the data into a consistent state, and, secondly, putting the conflict-causing application into the position to continue processing. Foresighted detection only requires the second step for resolution. Both steps can be further classified into *fully automatic* reactions, *semi-automatic* reactions, and *dynamic* reactions. Fully automatic means applying a general mechanism. For example, pessimistic concurrency control mechanisms used in database systems apply blocking, i. e., a transaction is made wait until the requested lock mode can be granted. This is an automatic reaction (after foresighted detection). Another possibility is letting the application decide, whether or not it wants to wait synchronously or asynchronously. Taking such application-specific a-priori specifications into account, is called semi-automatic resolution. The last kind of reactions are dynamic (i. e. manual) reactions, where the decision, how the inconsistency is to be eliminated or where the application will continue processing is dynamically taken by the user. Dynamic resolution seems to be especially reasonable in workflow environments, where it is extremely difficult to foresee all possible failure cases.

Some approaches towards activity management proposed in the literature use the term 'conflict prevention'. This mostly refers to the combination of foresighted detection with automatic reaction. In order to be more general, and, thereby, being able to provide more flexible/customizable conflict handling, we do not use the term 'prevention' and stick to the terms introduced above.

2.2 Classifying Known Approaches (Related Work)

In the subsection above, we analyzed conflict management and gave some intuitive examples of the most important ways of handling conflicts. This overview of the possibilities of conflict management will allow us to select the most appropriate ones for the CONCORD model. It is quite obvious that such a general view to conflict management is the foundation of finding the mechanisms which are best suited for a certain environment. In order to prove that we decomposed conflict management into the right components, we will now consider some approaches of conflict management chosen in well-known activity models proposed in the literature and will show how they fit into the 'conflict-management framework' introduced above.

Let us first consider the conventional transaction paradigm of database systems [GR93]. As far as checking of semantic integrity constraints is concerned, database systems following this paradigm usually use a subsequent detection and automatic resolution. Checking processing constraints (processing correctness here corresponds with serializability) depends on the mechanism exploited for concurrency control. A pessimistic two-phase locking protocol follows a foresighted detection (locks must be acquired) with automatic resolution (blocking until the lock can be granted or abort in the case of deadlock). Optimistic protocols exploit a subsequent detection (validation phase) with automatic resolution (abort).

In our opinion, the only universally applicable approach towards specifying application-specific correctness criteria (in contrast to serializability as an implicit correctness criterion) is realized in the 'Cooperative Transaction Hierarchies' model [NZ92]. Here, operations must be submitted and are to be checked against the specified correctness criteria (represented by finite, deterministic automata) whether or not execution can be granted in the current system state. This is an example for a foresighted detection with semi-automatic resolution. We have a semi-automatic resolution since the application is not blocked automatically but gets back control and can decide whether it wants to be informed as soon as the intended operation can be allowed due to the specified correctness criteria. Similar to other database-oriented approaches ([Ka95] gives a good overview) the 'Cooperative Transaction Hierarchies' approach requires users to derive (data-access) conflict specifications from knowledge about application semantics; another example is described in [Ba92] suggesting condition/action rules for recognition and resolution of data-access conflicts. This might overburden users and result in incorrect specifications. As we will see in the following subsection, we propose a stronger separation of 'technical' and 'semantic' aspects.

Considering a workflow system [GHS95, Le95] the execution of elementary actions (incorporated into workflows) is scheduled as specified. Thus, processing constraints are avoided, since it is assumed that specifications retain correctness of processing. This kind of application-specific avoidance shows the differences to foresighted detection. In case of foresighted detection, there is still a reaction necessary in the sense that the application must be put into the position to continue processing (see [NZ92]).

Another conflict management mechanism is used in CSCW tools [Gr94, GM94] supporting cooperation by notification. Often cooperating users are allowed to interleave actions, but as soon as an inconsistency occurs, the concerned cooperating person is notified in order to take the decision on how to react to this inconsistency. This kind of conflict management can be classified as subsequent detection with dynamic (or interactive) resolution. It is obvious that a dynamic resolution of subsequently detected conflicts can only be done at very application-specific levels of consistency. In our opinion, this is not acceptable, for example, at the level of database operations, since these have to retain basic consistency of the overall processing.

Although the approaches mentioned above exploit more or less fixed conflict management strategies, these do not necessarily suit best for all potential applications. Let us, for example, consider concurrency control (ACID transactions) in a highly interactive environment. If the granule of locking corresponds to user objects, it can also be adequate to let the user decide, whether or not he wants to wait until the locks acquired can be granted. It is quite obvious that such a dynamic conflict resolution strategy can only be exploited under the mentioned premises. Nevertheless, it may be a flexible solution in many situations. Such solutions become especially reasonable in the case of a layered system architecture. The next subsection will outline the meaning of ‘conflict management’ in those systems.

2.3 Conflict Management in Layered Architectures

In order to be able to deal with the inherent complexity of layered and even distributed systems, activity management same as conflict management have to be distributed to each subsystem. Hence, all activity-related and conflict-related decisions should be performed locally and only in exceptional cases, if a local treatment is not possible, the case has to be brought stepwise up to a system layer where a suitable conflict handler is able to cope with it automatically; otherwise, at the user level, manual resolution has to be performed. The interplay of activity and conflict management for the case of a layered system architecture is depicted in Fig. 1.

The obvious advantage of a layered system approach w.r.t. conflict management is that conflicts, which are hard to resolve at a certain layer must be treated at another layer. Some of the most important strategy characteristics of an efficient overall conflict management for a layered system are:

- Fine-grained and precise activity management should avoid conflicts as much as possible.
- Conflicts should be detected as early as possible and should be resolved at the same level whenever possible. This results in an automatic or semi-automatic conflict resolution as indicated in Fig. 1.
- Being unable to adequately resolving conflicts leads to passing on or even causing conflicts at the next upper layer.
- Of course, the highest conflict management instance is the user, who is aware of the global system state and has to resolve any remaining conflicts (dynamic resolution, see Fig. 1)

If conflict management is designed along these guidelines, the following desirable properties are conceivable for a particular layered application system:

- The scope of local conflict management is determined by the local system or the current local system state, respectively, being characterized as the already defined tuple (O_i, C_i, M_i, A_i, H_i), where the index i determines the system layer.
- Minimal layer interaction is guaranteed.
- Conflict resolution at a lower level contributes to conflict avoidance at a higher level.

• Conflict avoidance at a lower level either implicitly resolves the conflict or the conflict migrates to a higher layer, where a belated conflict resolution has to be applied.
• Conflict migration can be effectively used in order to move a conflict to the layer, where it can be handled most effectively.

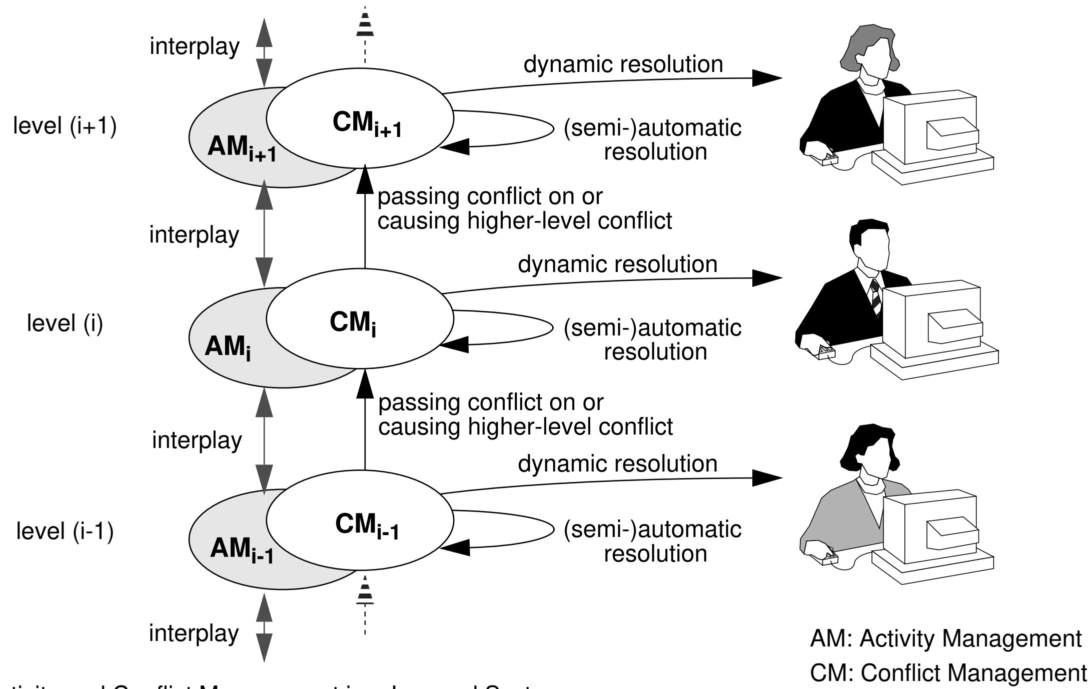

Fig.1: Activity and Conflict Management in a Layered System

Our CONCORD system refers to a layered system architecture that employs a layered activity management that collaborates with a layered conflict management as is also depicted in Fig. 1. In the following sections, we will prove why it is important to support the following features and how they are realized in CONCORD:

• Whenever possible, application-specific knowledge should be exploited to avoid conflicts.
• In the case that avoidance is not possible, foresighted detection of conflicts must be aimed at.
• Subsequent detection can only be tolerated, if an earlier detection is not possible or would cause unacceptable overhead.
• Conflict-resolution possibilities provided by a system should be very flexible.
• A layered system should pass (potential) conflicts to the layer they can be dealt with most effectively.

3. Overview of CONCORD

The CONCORD (CONtrolling COopeRation in Design environments) model [Ri94] captures the dynamics inherent to design processes. To reflect the spectrum of requirements, such as *hierarchical refinement, goal orientation, stepwise improvement and team orientation*, three different levels of abstraction are distinguished as roughly illustrated in Fig. 2.

3.1 Administration/Cooperation Level (AC Level)

At the highest level of abstraction, we reflect 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 key 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 constructed resembling (a hierarchy of) concurrently active tasks. All relationships between DAs essential for cooperation are explicitly modeled, thus capturing task-splitting (cooperation relationship type *delegation*), exchange of design data (cooperation relationship type *usage*), and negotiation of design goals (cooperation relationship type *negotiation*). The inherent integrity constraints and semantics of these cooperation relationships are enforced by a central system component, called *cooperation manager.*

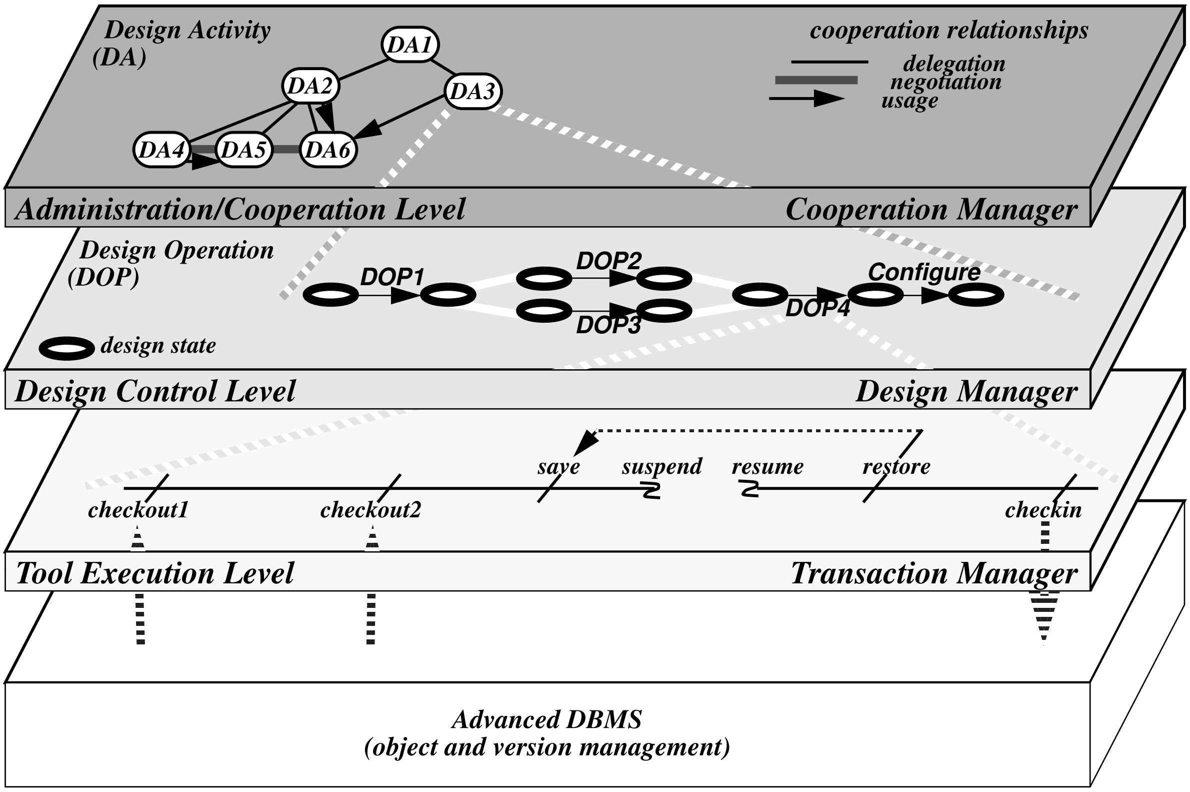

Fig. 2: Abstraction Levels of the CONCORD Model

3.2 Design-Control Level (DC Level)

Looking inside a DA reveals the DC level. There, the organization of the particular actions to be performed in order to fulfill a certain (partial) design task is the subject of consideration (*design flow*). At this level, Fig. 2 shows an execution plan (state/transition graph) of a particular design activity. It models the *control/data flow* among several design actions performed within a DA. Usually, these actions are design tool applications. The operational unit serving for the execution of a design tool is the *design operation (DOP*). In order to control the actions within the scope of a single DA, but without restricting the designers' creativity, flexible mechanisms for specifying the design flow for a DA are provided. The correctness of tool invocations is guaranteed by a system component, called *design manager*. The design manager also provides for recoverable design-flow executions that is needed for level-specific and isolated failure handling. 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 (object) states are captured by means of an object and version model.

3.3 Tool-Execution Level (TE Level)

From the viewpoint of the DBMS or data repository, a DOP is a long transaction (see Fig. 2). A DOP has the properties of conventional transactions. Because of long duration, it is internally structured by save/restore and suspend/resume facilities [Hä88] to be able to rollback the design at the application level and to continue the design work after breaks. A DOP processes design object versions in three steps. First, the input versions are checked out from the integrated data repository, and cached in an object buffer at the workstation for efficiency reasons. Second, the design data is mapped to in-memory storage structures tailored to the application needs. It is processed by one or more design tools. Third, the finally derived new versions are 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.

3.4 Design Data Repository

The three mentioned abstractional layers are provided on top of an object and version management system consisting of the advanced database management system PRIMA and an additional version-management layer [KS92]. The latter supports (the manipulation of) explicit versions of complex-structured objects.

4. Conflict Management in CONCORD

As outlined in Sect. 3, the CONCORD approach to both activity management and corresponding conflict management is different from those mentioned in Sect. 2.2 in that CONCORD refers to a comprehensive activity model that allows for straightforward mapping of all processing structures predominating in design processes. This is reached by integrating cooperation-control facilities and further mechanisms supporting design management and design-flow management with ACID transactions used as elementary building blocks. Along with this considerably enhanced activity model CONCORD supports a sophisticated and flexible conflict management that is capable to deal with all conflict scenarios mentioned earlier. In this section we elaborate the conflict management techniques best suited for a certain CONCORD layer. The discussion of the encompassing and integrated conflict management is given in Sect. 5.

4.1 (Data-) Model-inherent Aspects of Conflict Management

As already mentioned in Sect. 3.4, CONCORD relies on versioning. The 'object and version data model' provides explicit versions of complex objects and supports corresponding configuring. It is an inherent feature of versioning that concurrent transactions may derive different successors without getting a conflict, provided the considered version is allowed to have multiple successors in the underlying derivation structure. In this way, versioning provides a *general avoidance* of write/write conflicts. As already mentioned in Sect. 2.3, often avoidance at a certain layer demands for solving the conflict at a higher activity layer. Since (unrestricted) derivation structures allow arbitrary proliferation, there must be a kind of subsequent conflict resolution by merging versions or by selecting versions into configurations. The latter can be considered as container for 'important' versions and, thereby, establish a consistent composition of versions. Often merging and configuring cannot be reasonably done automatically; special application programs must be provided to do this task by taking application-specific semantics into account. This means that the price for conflict avoidance by versioning is the provision of merging and configuring tools, which are to be executed at the DC level, typically in an interactive mode.

4.2 TE-Level: Managing Data-Access Conflicts

Let us consider the TE-System, the bottom layer of the CONCORD system. Within the TE-System objects and versions can be manipulated by the generic operators of the 'object and version model'. DOPs are considered to be the processing units manipulating design data. To clarify the properties and assertions of DOPs, we, first of all, have to derive the underlying constraints asserting basic consistency of design data.

Processing Constraints

Let us first discuss processing conflicts. Since the data, DOPs of different DAs have access to, may highly overlap (due to cooperation), there are two kinds of concurrency. Firstly, parallel DOPs of the same DA may access the same (local) data concurrently. Secondly, DOPs of different DAs may access data which became accessible to these DAs due to cooperation. DOPs are supposed to be used to map design tools to system processes. In order to determine an adequate notion of correctness, we had to take into account, whether or not design tools do work cooperatively, and, if they do, whether or not the natural cooperative capabilities already provided by the versioning concepts are sufficient. After having examined VLSI design processes, mechanical engineering processes as well as software engineering processes, we came to the conclusion that tools do not directly cooperate. Furthermore, we realized that it is a natural way of processing that tools derive versions independently and that there are special tools for merging and configuring. This lead us to isolate DOPs, i. e. to exclude the well-known multi-user anomalies such as unrepeatable read [GR93]. To achieve the wanted degree of isolation, we developed the C^3-locking-protocol (C^3 stands for Concurrency Control in CONCORD) [Ri96] which is an extension of the well known two-phase locking protocol (2PL) [GR93]. The major requirements were the following. The protocol has to provide an adequate protection for versioned data. Due to long duration, abort cannot be used as a general mechanism for solving conflicts (deadlocks). Furthermore, the protocol could take the inherent properties of design processing into account, e. g. high interactiveness.

Version Lock Mode	Internal Effect	External Effect
VS (version read)	reading the version;	reading the version, deriving a successor version;
VD (version derive)	reading the version, deriving a successor version;	reading the version, deriving a successor version;
VDX (version exclusive derive)	reading the version, deriving a successor version;	reading the version;
VX (version exclusive)	reading/updating the version, deriving a successor version;	no rights;
VXC (version exclusive convertible)	reading the version;	no (further) rights;

Table 1 illustrates the lock modes obtainable for versions. The column ‘Internal Effect’ lists the rights of the lock holding transaction and the column ‘External Effect’ correspondingly lists the remaining rights of others. Considering a single version and a request of a transaction to derive a successor version, we had to foresee different mechanisms. Provided, the derivation graph structure, the version is embedded in, allows to derive multiple successors, the system grants a VD lock, giving concurrent transactions the possibility to also check this version out for derivation. If, on the other hand, the derivation structure does not allow multiple successors, e. g. in the case of a list structure, then the VDX mode must be given.

We also have to explain the two exclusive modes. VX is self-explaining. The VXC mode reminds at the Update lock mode introduced to prevent conversion deadlocks on hot spots [GR93] (in the original meaning it is granted instead of a shared lock). In the C^3-protocol, however, this mode is used differently, as we will see in the following. In our checkout/checkin scenario, manipulations are anyway carried out on a copy (stored within the object buffer) so that it is often appropriate to give the VXC mode (instead of an exclusive lock) and prevent from that point in time on further transactions from getting shared modes (VS, VD, VDX). So, there is a high probability that the initially active transactions holding shared locks will be committed till the considered transaction will get to its checkin step in which the VXC mode must be converted to VX.

Since usually the processing context of a design tool is known in advance and design flow can often be preplanned, we found it acceptable to introduce some kind of preclaiming [GR93], which we called pre-specification of access mode, because it is not completely equivalent to preclaiming as we will see. Thus, with each query the application specifies which access mode it wants to get on the query's result set. Fig. 3 shows the pre-specification clauses and the corresponding compatibilities of modes.

The major principles of the protocol are as follows. For each version in the query's result set, the lock manager checks whether or not the requested mode can be granted. If the requested mode is incompatible, the lock manager determines the highest possible mode (which could be granted on all elements) and offers it to the application. The application program now has the possibilities to accept the lower mode, to reject and get back control, or to wait synchronously or asynchronously. The drawback of this concept is that the application programmer has to deal with concurrency control aspects, but we think that this is restricted to an acceptable extent. For example, it can be appropriate to accept a derivation lock for a certain version instead of the originally requested update lock and not having to wait until other (usually long running) DOPs will terminate. In a similar way, VXC modes are handled. After an 'SELECT ... FOR UPDATE' request, the lock manager may offer a VXC lock, if there is a certain probability that the design tools which are currently holding incompatible, shared locks will be finished until the requesting tool gets to its checkin step.

We just want to add, how deadlocks are handled and which correctness criterion is achieved, since these aspects directly contribute to the topic of conflict management at this system layer. Deadlocks may occur, since we allow DOPs to initiate several checkout steps. Nevertheless, deadlocks are expected to be very infrequent, since they may only occur, if transactions decide to wait synchronously. Additionally, due to the pre-specification deadlocks can be detected very early, i. e. during the checkout phase, so that it is acceptable, to use abort as resolution. Since the manipulation phase is expected to be the most time-consuming phase within DOP processing, a reiteration of checkout is not too expensive.

lock held by a concurrent transaction
lock requested

		VS	VD	VDX	VXC	VX
SELECT VERSIONS [FOR READ]	VS	+	+	+	-	-
SELECT VERSIONS FOR DERIVATION	VD	+	+	*	-	-
	VDX	+	*	-	-	-
SELECT VERSIONS FOR UPDATE	VXC	+	+	+	-	-
	VX	-	-	-	-	-

Fig. 3: Compatibilities

Using the above mentioned principles within a two-phase locking protocol ensures at least 2.9 degree isolation (cf. [GR93], degree 3 is equivalent to serializability). This means that dirty reads, lost updates and unrepeatable reads cannot occur. Additional phantom protection would require to extend the lock granule.

Since phantom protection is hardly required in the design environment, C^3 does not provide phantom protection by default, but the application may switch to full serializability.

Having outlined the C^3-locking-protocol, we can now summarize the conflict management used at this system layer w.r.t. *processing constraints*. We have a *foresighted detection*, because of the pre-specification of access mode. That means, conflicts can be resolved before any operation has been executed. The *resolution* can be *automatic* (in the case that the request has to be rejected at all) or *semi-automatic* (in the case that the application programmer has foreseen reactions to offers from the lock manager) or even *dynamic* (in the case that the application programmer specified to pass the offer from the lock manager to the user and let him decide).

Data Constraints

Let us now turn to the handling of data conflicts. To do this, we, first of all, have to introduce semantic integrity conditions. The first class is called *general assertions* and is comparable with the well-known assertions of relational database systems [DD93]. The scope of general assertions is the entire data repository. This means that every DOP, no matter from which DA it has been initialized, has to fulfill the requirement not to invalidate a general assertion. Further classes of integrity conditions will be introduced in the following subsections. Thus, after the checkin of a DOP the transaction manager or its consistency manager must evaluate all general assertions. As soon as one of these conditions cannot be evaluated to true, a conflict occurs. Simply aborting the DOP would be a waste of work. Therefore, the control is given back to the application program (DOP), the conflict is reported, and the DOP may start an exception handling, with the goal to correct the inconsistency. This increases the chance to get to a successful termination.

Summarizing the conflict management aspects w.r.t. *data conflicts*, we can say that data conflicts must be detected subsequently, because the evaluation scope may also contain data, which may not have been in the DOP's object buffer. Nevertheless, flexible resolution mechanisms can be provided. This is supported by giving the DOP the chance to correct the inconsistency. We have a *semi-automatic* resolution, if the application programmer has programmed correction actions and we have a *dynamic* resolution, if the user is triggered to do correction actions interactively. An automatic reaction (abort of the DOP) must only be executed, if the DOPs exception handling was not successful; thus, an abort cannot be completely excluded in the case of a subsequent conflict detection, but is expected to occur very infrequent.

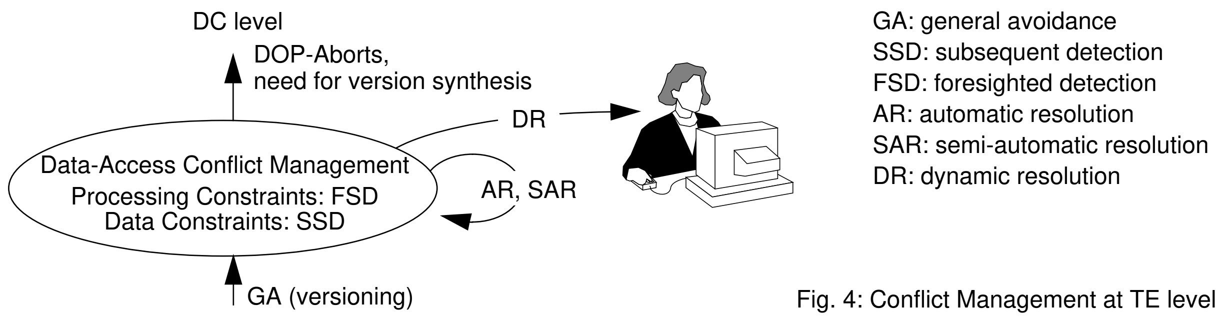

Fig. 4: Conflict Management at TE level

Summarizing Data-Access Conflict Management

Summarizing the overall conflict management at the TE level, we can say the following (cf. Fig. 4). We have to deal with ***data-access conflicts***. Data constraints can only be detected subsequently, processing constraints can be detected foresighted. In both cases, the full range of resolution mechanisms can be supported so that we have very flexible resolution possibilities helping to save as much work as possible. By these mechanisms, we succeed not only in flexibly resolving conflicts, but also in passing conflicts as few as possible to the next higher system layer which is the DC level. An unsuccessful termination (abort) of a DOP, however, can raise processing conflicts at the DC level, as we will see in the following subsection.

4.3 DC-Level: Managing Strategy Conflicts

At the DC level, we consider the internal structure of a single DA. Here only those objects, versions, and configurations are visible, which are relevant for data flow aspects. The set of operators is equivalent to the set of available design tools. To obtain an allowed history of tool applications, we, firstly, have to consider the set of constraints, especially the processing constraints.

Processing Constraints

The processing constraints to be observed at this layer are application-specific and, therefore, are specified by the user who created the DA. Fig. 5 shows an excerpt of a sample design-flow specification arranging the design steps of a particular DA (which is responsible for designing the body of a new car). Note that the user does not have to specify workflows in the language of the example. The CONCORD system provides a graphical specification tool guiding the user in creating design-flow specifications. The graphical specifications are internally mapped to the specification language. Nevertheless, we see in the example that the basic notion of design-flow specifications is the transition. A transition transfers design states into each other. A design state can be imagined as the set of currently accessible design data. A transition specification contains a pre- and a post-condition as well as an activity description. The conditions have the usual semantics. The activity description is more interesting. Here, we provide the possibilities of specifying design-tool invocations, sub-design-flow specifications or abstract activity descriptions. Nesting of design-flow specifications can also be used to control cooperative phases as we will see in the next subsection. An abstract activity description means just leaving it to the designer on how to fulfill this design step, i. e. letting him dynamically decide on which tools to execute in order to fulfill the post-condition. Besides all that, well-known concepts for control flow (e. g. the sequence used in the example of Fig. 5, see CONTROL-FLOW clause) and data flow are supported, which we do not want to discuss in this paper.

```
DESIGNFLOW_DESCRIPTION Body_Design
    INPUT_PARAMETERS (in: inf_data)
    OUTPUT_PARAMETERS (out: body_data)
    OUTPUT_BINDINGS (out:=S3.body)
    TRANSITIONS (
      (T1    (TRUE)
             (Determine_Frame_Data
             IN (in := START.in) OUT (Frame := out))
             (Frame.state = verified))
      (T2    ...)
      ...
    CONTROL_FLOW (S1->(T1, S2)->(T2, S3)-> ... );
```

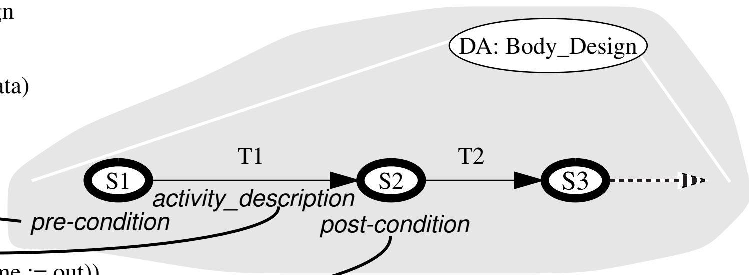

Fig. 5: Excerpt of a Sample Workflow Specification

Now, we can summarize the conflict management principles exploited at the DC level w.r.t. *processing constraints*. The design-flow specifications and the corresponding scheduling of design steps establishes an *application-specific conflict avoidance*, since correctness of processing here means following the specifications and this is ensured by the design-manager component. Note that this simple comprehension of correctness is based on the fact that we are just considering local design data, i. e. data belonging to the scope of the considered DA. The notion of correctness w.r.t. cooperation between DAs will be detailed in Sect. 4.4.

Data Constraints

Let us now turn to data constraints relevant at this level. In the previous subsection, we introduced general assertions which are to be asserted by every DOP. Pertaining the DC level, we have to add another class of semantic integrity constraints. This class is called *DA-specific assertions* and is related to the local data scope of a DA. This implies that DOPs initiated by a certain DA have to fulfill the DA-specific assertions of this DA. A DA-specific assertion may be active during the entire life-time of a DA or may be switched on or off. For example, it may be reasonable to use a DA-specific assertion to enforce that already reached features (properties the resulting design object should have) will not be invalidated during further processing.

Although, DA-specific assertions are associated with the DC level, the ckecking is delegated to the TE level. Thus, at checkin of a DOP the transaction manager or its consistency manager not only has to check general assertions but also has to communicate with the design manager, which DA-specific assertions and which post-conditions must be enforced. Only, if all these assertions can be evaluated to true, the DOP can be committed. The handling of pre-conditions must be characterized a bit differently. Pre-conditions express prerequisites of actions. In this way, pre-conditions and their checking establish a *foresighted detection* and the *reaction again can be either automatic, semi-automatic or dynamic,* depending on how strict the corresponding design-flow specifications are.

Summarizing Strategy-Conflict Management

The discussions in this section show that we are dealing with ***strategy conflicts*** at this level (see Fig. 6). *Processing conflicts are avoided due to application-specific design-flow specifications,* and *data constraints are partially detected foresighted (pre-conditions), partially subsequent (post-conditions, DA-specific assertions), and can be resolved flexibly*. Conflicts caused at the TE level and handled at the DC level are break-offs of DOPs, e. g. if exception handling is not successful. This is expected to be very infrequent, because of the flexible resolution possibilities, but can sometimes not be bypassed. Thus, at the DC level a resolution must be found, which is in accordance with the design-flow specification, e. g. retrying a design-flow branch or a single tool execution with changed input data.

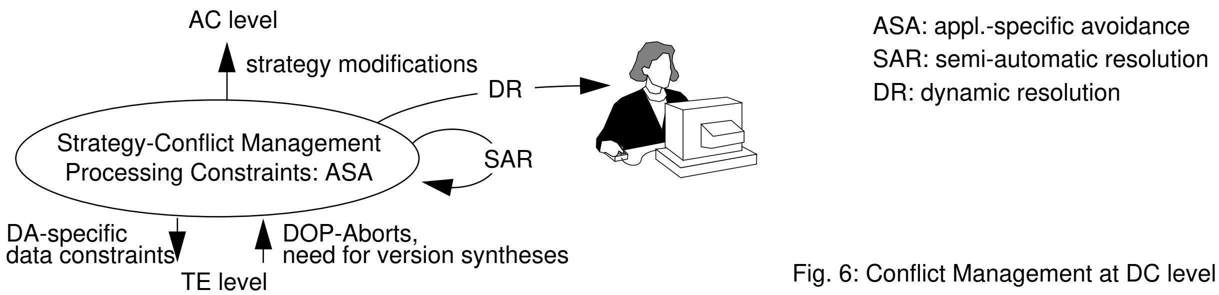

Fig. 6: Conflict Management at DC level

4.4 AC-Level: Managing Cooperation Conflicts

At the highest activity layer, we consider the AC-System manipulating objects, versions and configurations which are subjected to cooperation. This includes preliminary data which is exchanged between cooperatively working DAs (usage) as well as final data w.r.t. a certain DA which has fulfilled its task and passes the result back to the superordinated DA (delegation). Thus, the activity units are the (sub-)DAs incorporated into a design process and the operators are the corresponding design-flow types or the incorporated tool-invocations, respectively. The corresponding design flows may be interleaved or may access cooperation data according to application-specific correctness criteria. Constraints may be divided into cooperation-processing constraints and cooperation-data constraints, together restricting the set of allowed histories of tool applications.

Processing Constraints

Let us first consider the different types of cooperation supported in CONCORD. Although, cooperation specifications logically belong to the AC-level, they are embedded into design-flow specifications. Due to simplicity of specifications we used the same language. This language supports the specification of explicit as well as implicit cooperation. *Explicit cooperation* relies on operations which can be compared with operations on access rights as known from SQL2 [DD93]. Remember that all data which is created or derived by design-tool applications is, first of all, inserted into the local scope of the corresponding DA and can only be made accessible for other DAs due to cooperation. Now, the operations *permit* or *transmit* can be used to cooperatively exchange preliminary design data. Permit grants a certain access right (read, derive or update) on a certain design-object state to a cooperating DA. Rights given by the permit operation can be withdrawn again by issuing the *revoke* operation. The operation *transmit* transfers ownership from one DA to another one; this can only be undone by issuing the transmit operation again (into the opposite direction). The usage of these operations can be allowed between certain DAs in certain design phases or can explicitly be pre-planned. Fig. 7 gives, at the left-hand side, a sample specification of a pre-planned, explicit cooperation phase; the illustration at the right-hand side shows how this specification can be embedded into the design-flow specifications of the involved DAs.

The example shows how cooperation between DAs can be pre-planned as a sub-design-flow specification appearing in each of the design-flow specifications of the two DAs involved. After having reached design state S1 (which is associated with state K2), DA1 permits access on the designed engine to DA2. Then DA2 uses this information to design the transmission. Afterwards the access rights are revoked. Besides this pre-planned invocation of cooperation operations, there is the possibility to allow the dynamic usage between certain DAs in certain design phases. Suppose, the two DAs of the example both have a sub-design-flow with abstract (open) activities. Within the specifications of these sub-design-flows it can be specified that the corresponding designers are allowed to issue cooperation operations dynamically.

```
EXPLICIT_COOPERATION_DESCRIPTION Phase_1
  INVOLVED ARE (DA1, DA2)
  OUTPUT_PARAMS (out: transmission_data)
  OUTPUT_BINDINGS (out:=S3.transmission)
  TRANSITIONS (
  (T1   (TRUE) DA1: PERMIT (DA1, DA2, K2.engine, R) (TRUE))
  (T2   (TRUE)
        DA2:        (Transmission_Design,
        INPUTS (    in1:=DA2.START.in,
                    in2:=DA1.K2.engine)
        OUTPUTS ( transmission:=out))
        (transmission.state=verified))
  (T3   (TRUE) DA1: REVOKE (DA1, DA2, K2.engine, r)) (TRUE)))
  CONTROL_FLOW (S1->(T1, S2)->(T2, S3)->(T3, S4));
```

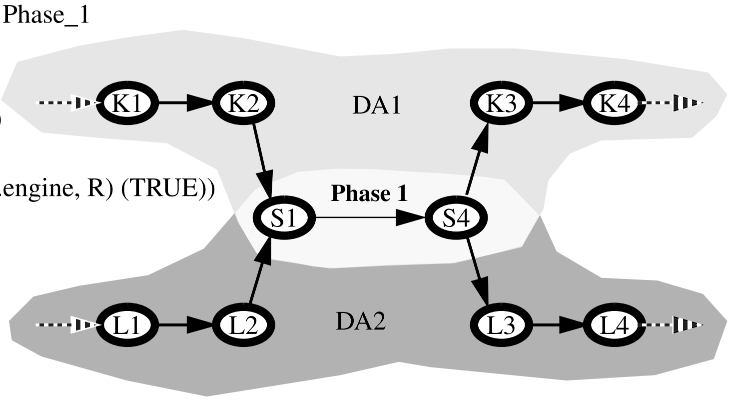

Fig. 7: Sample Specification for Explicit, Pre-planned Cooperation

The basic idea of *implicit cooperation* is that there are no local data scopes of the cooperating DAs, but there are object pools the DAs are concurrently working on by applying design tools. Correctness of processing can only be enforced by specifying, which sequences of design-tool applications are correct and which are not. For that purpose, we exploit the mechanism of finite, deterministic automata, known from the 'Cooperative Transaction Hierarchies Approach' [NZ92]. A sample specification is illustrated in Fig. 8.

We see the specification of an implicit cooperation phase, which can be imagined as a sub-design-flow occurring in the design flows of the involved DAs. The object pool, being the set of data the cooperative phase is working on is initialized in the INPUT_PARAMS clause and the correctness criteria is given by the automaton A1. Due to simplicity we use just one automaton in the example. The mechanism still holds for a set of automata. The basic idea is the following. Suppose we are in the initial state S1. Now, the involved DAs may issue design-tool applications on the data in the object-pool. The final state of the automaton (S2) can only be reached by verifying the frame_data (upper transition). The design phase can only be left when the automaton is in the final state. But if, after having verified the frame_data, a DA again modifies the frame_data, and, thereby, makes S1 being the current state (lower transition), again the verify tool must be invoked, before the (cooperative) design phase can be finished. Thus, the automaton enforces the last step of the phase to be the verification step. In a similar way, more sophisticated patterns of cooperation can be specified. We just wanted to explain the basic idea.

This finishes our outline of the possibilities in specifying user-defined correctness criteria related to special usage relationships. Delegation and Negotiation relationships follow pre-defined protocols which do not have to be adapted to application-specific needs. Since these protocols are intuitive, we do not detail them.

```
IMPLICIT_COOPERATION_DESCRIPTION Phase_2
    INVOLVED ARE (DA1, DA2, DA3)
    INPUT_PARAMS (in1: engine_frame_data, in2: transmission_frame_data, in3: body_frame_data)
    OUTPUT_PARAMS (out1: engnine_frame_data, out2: transmission_frame_data, out3: body_frame_data)
    AUTOMATA (A1);

FINITE_STATE_MACHINE A1
    STATES (S1, S2)
    TRANSITIONS (
        (S1, S2, ANY, Verify_Frame_Data, ACCEPT)
        (S2, S1, ANY, NOT Verify_Frame_Data, ACCEPT))
    START_STATE S1
    END_STATES (S2);
```

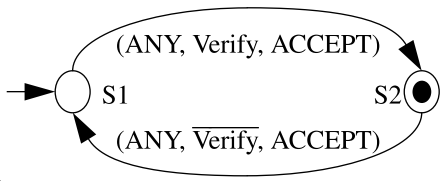

Fig. 8: Sample Specification for Implicit Cooperation

Now we can summarize the conflict-management principles exploited at the AC level w.r.t. (cooperation) *processing constraints*. The explicit, pre-planned cooperation establishes an *application-specific avoidance*, because similar to the DC level the notion of correctness is equivalent to accordance of processing with specifications. Implicit cooperation follows a *foresighted detection* strategy, implemented by the automata approach. The corresponding resolution strategy can only be *dynamic*, because implicit cooperation is assumed to not follow flow specifications (tools are dynamically invoked by designers). Explicit, dynamic cooperation is, as mentioned above, just the permission to use the (generic) cooperation operations (request, permit, revoke, transmit) between certain DAs in certain design phases. These operations leave it completely to the designer to distribute preliminary data and to resolve possibly occurring conflicts. Thus, these operations rely on a *dynamic resolution* (e. g. by revoking an earlier given permission). On the other hand, explicit cooperation operations provide a *basic (automatic) conflict detection*. As soon as cooperatively exchanged data (which has really been used within DOPs of cooperating DAs) is withdrawn or is updated (in place), the using DA, or the corresponding designer, respectively, is notified and, thereby, put into the position to decide how to react. Thus, detection in the case of explicit, dynamic cooperation means recognizing withdrawals of used data; the basis for the dynamic resolution is notification.

Data Constraints

Data constraints are crucial w.r.t. cooperation as we will see in the following. As already mentioned, all DA-related design data is associated with certain access rights. Ownership is registered as well as permissions for read, write and derivation access. Ownership automatically includes all permissions and only the owner is allowed to delete design data. The operation permit grants read, write or derivation rights; the operation transmit transfers ownership but leaves the other rights. Thus, with using these operations, maybe in dynamic or pre-planned fashion, the data scope of DAs can be changed dynamically. Each DA, however, may define its own DA-specific integrity constraints. This raises the problem that permissions could allow a cooperating DA to change a design-object state in a way invalidating an assertion of the owning DA. In this way, inconsistencies may occur. Such inconsistencies are prevented as follows. When a DOP of a cooperating DA accesses data which is subjected to cooperation, DA-specific constraints of the owning DA are checked within the scope of the owning DA (during commit). This is the basic idea implementing a subsequent detection with flexible resolution possibilities (in the same way as already discussed for enforcing general assertions at the TE level and DA-specific assertions at the DC level).

Generally, all DA-specific assertions are evaluated against the DA's data scope containing the data owned by this DA and, additionally, the data made accessible for this DA by cooperating DAs. This also implies that all operations changing the scope of a DA (permit, transmit, delete, ...) can only be accepted, if the DA-specific assertions stay valid. Regretfully, we cannot deepen this aspect due to space restrictions.

Summarizing Cooperation-Conflict Management

Let us now conclude the conflict management mechanisms used at the AC level to deal with ***cooperation conflicts*** (see Fig. 9). Similar to mechanisms used at the lower levels, *data conflicts* (conflicts occurring w.r.t. data constraints) are handled by *subsequent detection and flexible resolution*. Processing conflicts, on the other hand, are handled by *application-specific avoidance (explicit, pre-planned cooperation)* or by *foresighted detection and flexible resolution (implicit cooperation)*. So, also at this level, the, as we think, most flexible mechanisms are provided. This is true even in the case of *explicit, dynamic cooperation, where, pertaining withdrawals, only subsequent detection with dynamic resolution* is possible.

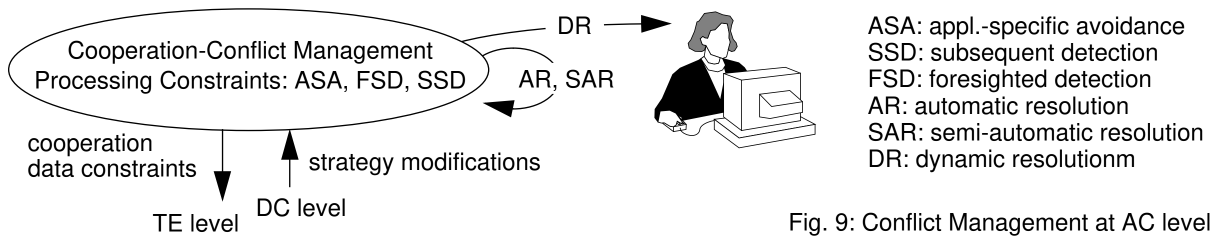

Fig. 9: Conflict Management at AC level

5. Integrated Conflict Management

After having detailed the conflict management principles exploited at the various system layers of the CONCORD system in the previous sections, we want to give an integrated view to the CONCORD conflict management in this section. In Fig. 10, overall CONCORD conflict management is illustrated. We see that the transaction manager is responsible for the handling of data-access conflicts, the design manager has to handle strategy conflicts and the cooperation manager has to contribute in resolving cooperation conflicts.

The arrows in Fig. 10 indicate resolution mechanisms. In CONCORD the user (designer) exclusively communicates with the design manager associated with the DA he is responsible for. Each time a dynamic conflict resolution becomes necessary, the design manager functions as a mediator between designer and conflict-detecting component. Besides dynamic resolution, all levels support the possibilities of automatic and semi-automatic resolution. Thus, CONCORD provides highest possible flexibility in conflict resolution. The conflict management ellipses contain the exploited mechanisms for avoidance or detection w.r.t processing constraints. We see that CONCORD mainly exploits application-specific avoidance or at the more technical TE layer foresighted detection. The AC layer provides the whole spectrum from subsequent detection over foresighted detection till application-specific avoidance. Here, we have to take into account that subsequent detection is only used in the case of dynamic (explicit) cooperation and that this kind of cooperation does not allow an earlier detection or avoidance.

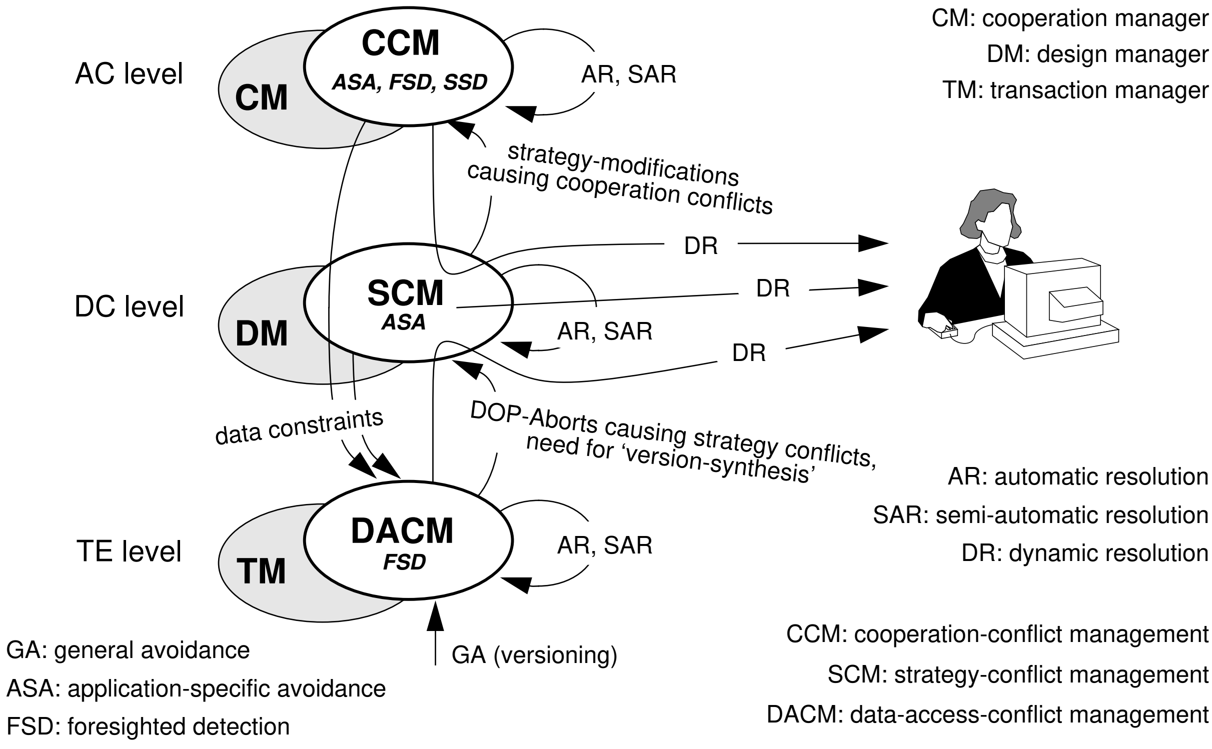

Fig. 10: Integrated Conflict Management in CONCORD

Conflict handling w.r.t. data constraints brings us to the possibility to move conflict handling to another layer, where it can be done more appropriately. The evaluation of general assertions (assertions associated with the TE level which are to be ensured by every DOP, no matter to which DA it belongs), design-flow conditions (DA-specific assertions, which are to be ensured by each DOP belonging to a certain design phase of the corresponding DA) as well as cooperation conditions (assertions which must be checked in the scope of the owning DA when cooperatively exchanged data has been manipulated) is shifted to the TE level, where it is carried out during DOP commit. Thus, 'subsequent' means 'after the actual database operation at the end of the corresponding transaction'. Regarding the underlying object-buffer concept for client-based design-data manipulations and the typically highly meshed structure of design data complex assertions cannot effectively be checked before checkin. So, we found ourselves forced to use subsequent detection which is acceptable together with flexible resolution mechanisms.

Other conflicts are shifted upward. For example, the general avoidance achieved by versioning requires that adequate tools must be executed (from the DC level) to provide an adequate merging or configuring of versions to reach meaningful design-object states. Aborting a DOP (note that this is expected to happen very infrequently in CONCORD but cannot be excluded completely) causes a conflict at the DC level. The design-flow management system may react by retrying the execution; but this can only be successful, if no semantic error was the reason for the abort, otherwise the abort can occur repeatably. If a retry is not successful, the system must trigger designer interaction for dynamic (i.e. interactive) resolution. He may change the input data or may step back to an earlier design state and decide for a different processing, e. g. choosing another branch, as far as the design-flow specification allows these changes. Stepping back in the design history to an earlier state may demand for withdrawals of cooperation data and, thereby, cause a conflict to be handled at the AC level by notification and dynamic resolution.

6. Conclusions

In the discussions of this paper, we have clearly shown that a conflict model and its corresponding conflict management are among the vital constituents of each activity model that is meant to realize a cooperative processing model as is needed in collaborative design environments and CAD frameworks.

Conflict management in CONCORD is designed as an integral part of the CONCORD activity model. The outstanding properties of CONCORD's conflict management can be described as follows:

- The conflict model handles data-access conflicts, strategy conflicts as well as cooperation conflicts.
- Conflict management is aimed at primarily avoiding conflicts (as far as processing is not restricted), or by detecting and resolving them as early as possible. Hence, CONCORD uses foresighted detection at the more technical TE level and application-specific avoidance as central mechanism at the higher levels.
- The conflict resolution is designed as flexible as possible. Therefore, CONCORD provides automatic and semi-automatic resolution besides dynamic resolution.
- Especially, dynamic resolution provides the highest degree of flexibility, since the user/designer provides a resolution by means of dynamic interaction with the design activity. Dynamic resolution is tractable whenever the kind of conflict and the concerned resources fit into the mental model of the user, i. e. are not too technical. We think that this is true in many situations in collaborative design environments, since often data granules accessed by DOPs correspond to design-object states known to the user.

The discussions showed that both the scope and the limits of conflict management are determined by the capabilities of activity management. Furtheron, in order to adapt and to optimize the conflict management for a layered system architecture (as in the case in CONCORD), conflict management same as activity management has to be distributed to each system layer. Thereby, the following guidelines turned out to be overly important w.r.t. overall efficiency:

- All conflict-related (as well as activity-related) decisions should be performed locally.
- If a local treatment is not possible, the case has to be brought stepwise up to a system layer where a suitable conflict handler is able to cope with it preferably in an automatic manner.
- Otherwise, dynamic (i.e. manual) resolution has to be performed at the user level.
- A layered system should pass (potential) conflicts to the layer they can be dealt with most effectively.

So far, the capabilities of the CONCORD conflict model and conflict management have been worked out, discussed, and documented. In the future, we will concentrate on both, implementation techniques for layered conflict management as well as on completing our generic conflict model towards a conflict management framework. The latter is (as part of the activity-management framework) a vital component of any CAD framework. Additionally, a conflict-management framework is (same as the underlying activity-management framework) customizable and, thereby, can be adapted to the requirements and features of a particular design environment. Our next efforts w.r.t. implementation techniques for layered conflict management refer to strategies and techniques for the integration of conflict management and activity management.

Literature

Bo95 ten Bosch, O.: Design Flow Management in CAD Frameworks, PhD Thesis, Delft University of Technology, The Netherlands, 1995.

Ba92 Barghouti, N.S.: Concurrency Control in Rule-Based Software Development Environments, Ph.D. thesis, Columbia University, New York, 1992.

BK91 Barghouti, N.S., Kaiser, G.E.: Concurrency Control in Advanced Database Applications, ACM Computing Surveys, Vol. 23, No. 3, September 1991.

Br93b Brettschneider, F.: A Process Model for Design Flow Management and Planning, Fortschritt-Berichte VDI, Reihe 9; Elektronik, Nr, 157, VDI-Verlag, Düsseldorf, 1993.

DD93 Date, C., Darwen, H.: A Guide to the SQL Standard - A user's guide to the standard relational language SQL (3rd Edition), Addison-Wesley Publishing Company, 1993.

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

GHS95 Georgakopoulos, D., Hornick, M., Sheth, A.: An Overview of Workflow Management From Process Modeling to Infrastructure for Automation, Journal on Distributed and Parallel Database Systems, 3(2), April 1995.

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

GM94 Greenberg, S., Marwood, D.: Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface, in: Proc. ACM 1994 Conference on Computer Supported Cooperative Work, Chape Hill, NC, 1994, pp. 207-218.

Gr94 Grudin, J.: Computer-Supported Cooperative Work: History and Focus, IEEE Computer, May 1994, Vol. 27, No. 5, pp. 19-26.

Hä88 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.

Ha90 Harrison, D., Newton, R., Spickelmier, R., Barnes, T.: Electronic CAD Frameworks, Proc. of the IEEE, Vol. 78, No. 2, Febr. 1990, pp. 393-417.

Ka95 Kaiser, G.E.: Cooperative Transactions for Multiuser Environments, in: Kim, W. (ed.), Modern Database Systems - The Object Model, Interoperability, and Beyond, Addison-Wesley, New York, 1995, pp. 409-433.

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

Le95 Leymann, F.: Supporting Business Transactions via Prtial Backward Recovery in Workflow Management Systems, G. Lausen (Hrsg.) Datenbanksysteme in Büro, Technik und Wissenschaft, Springer, 1995.

MHR96 Mitschang, B., Härder, T., Ritter, N.: Design Management in CONCORD: Combining Transaction Management, Workflow Management, and Cooperation Control, Proc. 6th Int. Workshop on Research Issues in Data Engineering: Interoperability of Nontraditional Database Systems (RIDE-NDS), New Orleans, 1996.

NZ92 Nodine, M.H., Zdonik, B.: Cooperative Transaction Hierarchies: Transaction Support for Design Applications; VLDB Journal 1, 1992, pp. 41-80.

Ri94 Ritter, N., Mitschang, B., Härder, T., Gesmann, M., Schöning, H.: Capturing Design Dynamics - The CONCORD Approach; Proc. 10th Int. IEEE Data Engineering Conference, Houston, Texas, 1994, pp. 440-451.

Ri96 Ritter, N.: The C^3-Locking-Protocol - A Concurrency Control Mechanism for Design Environments, in: Proc. STAK'96, Softwaretechnik in Automatisierung und Kommunikation, Rechnergestützte Teamarbeit, Munich, Germany, 1996.

SSZ94 Simon, R., Sclabassi, R., Znati, T.: Communication Control in Computer Supported Cooperative Work Systems, in: Proc. ACM 1994 Conference on Computer Supported Cooperative Work, Chapel Hill, NC, 1994, pp. 311-321.