GROUP-AUTHORING IN CONCORD
-
A DB-BASED APPROACH

Norbert Ritter

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

Keywords: Groupware, Collaborative Editing, Cooperation, Transactions, Concurrency Control.

ABSTRACT

CONCORD provides processing support for concurrent design/engineering. As an extension of the well known transaction paradigm used in database systems, design transactions, designflows as well as cooperation between designers are supported. In this paper, we argue that group authoring and group editing can also be supported by CONCORD; a corresponding group editor will be presented. We show that the isolation property of conventional transactions is not inherently contradictory to cooperation. Due to consistency reasons, isolation is still necessary (at lower levels) and, therefore, can be considered as the foundation of cooperation, provided activities are adequately structured and suitable mechanisms are used to enforce isolation. Our group editor CGE proves that both, adequate working in the text document (in the sense that the processing principles visible at the interface fit into the user's mental model) as well as adequate cooperation (in the sense that users may exchange intermediate results) can be reached under exploitation of isolated transactions as a basic means.

INTRODUCTION

Motivation

The CONCORD processing model is a database-oriented processing model for design applications and has been proposed for and evaluated in design applications such as VLSI design, mechanical design and software engineering. It exploits the well-known transaction paradigm as a basic means for consistency maintenance. At the first glance, this might seem contradictory to cooperation requirements, but, as we will see, together with an adequate structuring of processing and adapted mechanisms realizing the ACID properties of transactions (ACID stands for atomicity, consistency, isolation and durability [9]) cooperation can be supported appropriately. In this paper, we show that the basic principles of the CONCORCD processing model can also be effectively applied in group-authoring applications, which are typical CSCW [8] applications.

In [11] the major requirements for a group editor are summarized. Concerning individual work a user may wish, on one hand, to be free of technical access restrictions, and, on the other hand, to be supported by private areas, adjustable granularity, and the freedom of making updates public at any time. We think that, due to consistency maintenance, we cannot afford excluding technical protocols completely. Using social protocols exclusively and, thereby, degrading the system to a conflict detector without resolution (or avoidance) capabilities may too often provoke users to spend more time in resolving recently caused conflicts and inconsistencies than in working productively and forward oriented. Thus, technical protocols are acceptable, as long as they are 'compatible' with social protocols. Consequently, we claim that technical protocols can effectively be exploited, as far as they allow natural processing and do not exhibit bothering effects at the interface.

Related Work and Overview of the Paper

Approaches towards support of collaborative writing proposed in literature can be divided into two major groups. First, there are systems supporting synchronous editing, e. g. GROVE [3, 2], DistEdit [10], CAVEDRAW [13], PREP [16]. Supporting synchronous editing means providing changes to cooperating team partners in real-time. The second group contains asynchronous writing tools, such as DUPLEX [18], PREP [15] and MESSIE [23].

As stated in [11], especially the synchronous tools are not frequently used, because they do not reflect the natural way of cooperative writing adequately. We think that there is definitely a need for WYSIWIS ('What You See is What I See'), but that does not mean that users want to modify overlapping text fragments simultaneously and it, furthermore, does not mean that WYSIWIS is required in all phases of a document production process or during all modifications on all parts of the document.

Asynchronous tools have primarily been proposed to tackle the problems of large scale environments being highly distributed, heterogeneous and having to care for 'secure' net communication and especially fault tolerance. Asynchronous approaches can be further classified in those exploiting a central data repository, e. g. PREP [15], and those exploiting replication, e. g. IRIS [11], CES [7], PREP [16].

Our approach exploits a central database to manage document data and process-management meta-information. It integrates facilities for synchronous as well as asynchronous cooperation. Synchronous cooperation, however, is kind of restricted, since there are no simultaneous write operations. Nevertheless, there can be a synchronous transfer of actions to cooperating users, provided all involved users agree (no privacy violation) and 'play' roles allowing at least read access to the corresponding document part.

Our major concern is concurrency control and corresponding document data consistency. Collaborative editing literature shows that there are three main approaches used in the corresponding systems. First, using a technical concurrency-control protocol, as known from database systems. Second, defining responsibilities of persons for document parts via roles users may play. This reminds at access-control concepts also used in conventional database systems. Third, exploiting social protocols, i. e. relying on the assumption that users work together in a way preventing conflicts.

Some systems, e. g. DUPLEX [18], provide several of these principles, forcing users to care for concurrency-control problems, since they have to chose appropriate protocols for the work on particular parts of the document. Technical protocols [1, 19], e. g. pessimistic concurrency-control protocols, are usually implicit. The problem with these mechanisms is that they, if used in highly interactive environments as group editing, lead to bothering effects at the interface [6]. For example, users do not tolerate arbitrary wait situations, which may occur, when the system employs implicit blocking due to serialize actions or to ensure serializability [5], respectively. In our opinion, these problems are particular problems of the used mechanisms, not of the transaction properties, they ensure. Isolated transactions, in our opinion, can be used, even in group-editing applications, if they (1) capsulate exactly those data-manipulating operations needed to execute a semantically coherent modification step and (2) isolate exactly that portion of data semantically concerned by the modification (e.g. flexible lock granules) and (3) conflict resolution fits into the mental model of the user, i. e. corresponds to some conflict resolution actions which can also be imagined to be used within social protocols.

In this paper, we want to describe such an approach. It resulted from the validation of the database oriented processing model CONCORD, developed for technical design applications, in the group-editing scenario. After this introduction, Sect. 2 will give an overview of CONCORD. Sect. 3 and 4 introduce the concepts of the CONCORD Group-Editor CGE and outline its implementation. Sect. 5 gives conclusions and an outlook to future work.

CONCORD PRINCIPLES

The CONCORD (CONtrolling COopeRation in Design environments) model [20] captures the dynamics inherent to design processes. Similarly to [22] CONCORD aims at combining principles of transaction management, workflow management [4] and cooperation control [14] in order to support both, well structured as well as less structured cooperative applications. To reflect the spectrum of requirements different levels of abstraction are distinguished as roughly illustrated in Fig. 1.

Design-Data Management

W.r.t. engineering applications the CONCORD processing model relies on a version model (OVDM [12]) providing means for the management and manipulation of explicit complex-object versions. Especially in applications such as VLSI design or mechanical design, typically complex-structured design data must be managed and versions as well as configurations must be supported adequately. Although the OVDM has primarily been developed to handle complex structured and versioned data, it is also capable to handle version-free and flat data structures, so that the relational model can be seen as being a part of the OVDM's functionality.

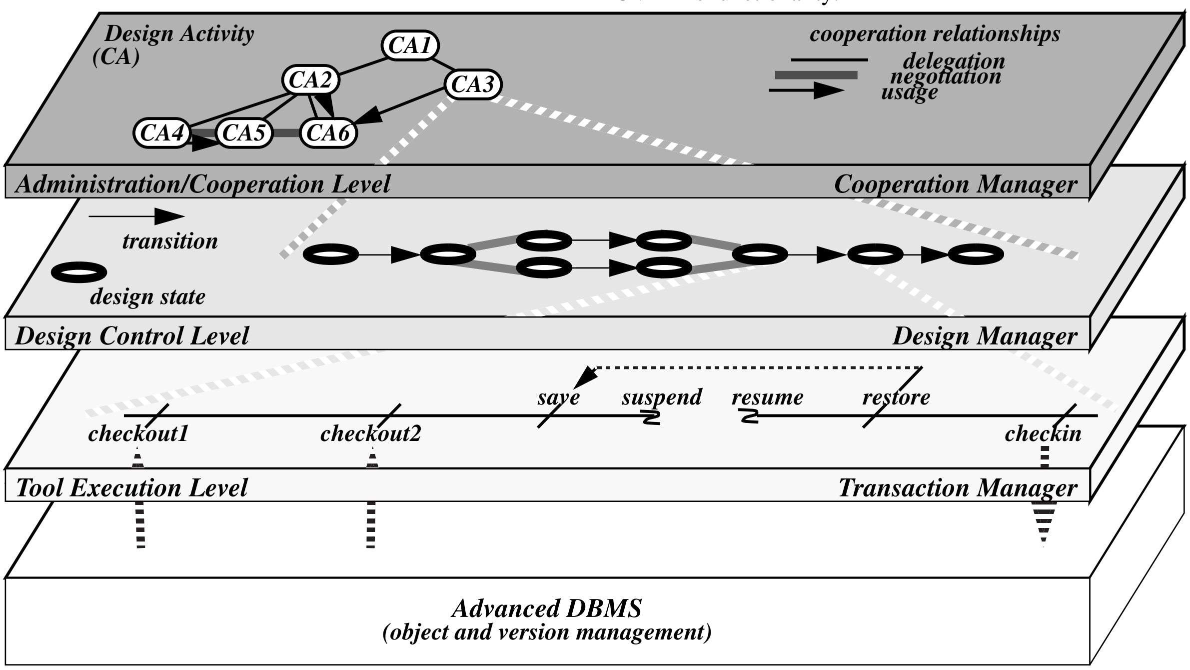


Fig. 1: Abstraction Levels of the CONCORD Model

In the group-authoring context, we decided to consider document data to be unversioned. The reason is that versioning is contrary to flexible locking granules, which, as we will see, is a major foundation of our approach. Furthermore, this avoids the problem of semantic merging of versions, which cannot be supported by automatic mechanisms.

Tool-Execution Level (TE Level)

CONCORD provides long ACID transactions (see Fig. 1). In this context, *long* does not necessarily mean long duration, but refers to the concept of checking out data from the server into an object buffer at the workstation, processing it there, and, eventually, propagating back the modified data within a final checkin step. Schema-consistency and persistency is guaranteed by a central system component, called *transaction manager*. It is also responsible for the isolated execution and for recoverability of transactions, which are indispensable properties for a level-specific failure handling.

Design-Control Level (DC Level)

The transactions mentioned above are used to map design-tool applications to system processes. The DC level incorporates a design-flow model allowing the flexible pre-specification and scheduling of design-flows as well as of ad-hoc design-flows. The design-flow model is state/transition based, where the states stand for design-object states and the transitions, in the simplest case, are design-tool applications or transactions, respectively. Since we think that group-editing applications do not require workflow capabilities, we do not detail this layer.

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,* also called *cooperative activity (CA)*. A CA is the operational unit representing a particular task or subtask. During the design process, a *CA hierarchy* can be dynamically constructed resembling (a hierarchy of) concurrently active tasks. All relationships between CAs essential for cooperation are explicitly modeled, thus capturing task-splitting (cooperation relationship type *delegation*), negotiation of design goals (cooperation relationship type *negotiation*), and exchange of design data (cooperation relationship type *usage*). The inherent integrity constraints and semantics of cooperation relationships are enforced by a central system component, called *cooperation manager.* Especially the protocols which are associated with usage relationships are interesting in the context of database-supported cooperation. Explicit cooperation relies on the fact that each CA is associated with local design data. Between CAs access rights can be explicitly and dynamically granted. For technical applications, where design-flows can be pre-planned, explicit cooperating can also be pre-planned. A third kind of cooperation, implicit cooperation, exploits the mechanisms proposed in [17] to control design-tool applications issued by cooperating CAs on shared object pools. Since this way of cooperation is not relevant for group editing, we do not detail it.

BASIC CGE FEATURES

CGE follows the mentioned CONCORD principles. It is not a tool designed to be used within the CONCORD activity framework; CGE directly implements those parts of the three (AC-, DC-, TE-) subsystems, which we found to be relevant for group authoring. In the following, we discuss these features. During developing CGE processing-control, we emphasized fulfilling the following requirements:

- in order to provoke as few conflicts as possible, manipulation operations must not access document data (text), which is not semantically affected by the modification;
- there must not be implicit blocking of user activities;
- conflicts must be detected early, i. e. foresighted, and conflict resolution must be very flexible.

Close Cooperation

CGE allows to create single-user CAs as well as multi-user CAs. The principles of working within a CA are:

- users dynamically invoke text-manipulating functions provided by CGE;
- each function, e. g. reading text or manipulating a part of the text, is a database transaction, i. e., read/write operations are isolated against each other;
- results of manipulation steps can be directly accessed by users associated with the same CA;
- the current state of a text (segment) associated with the CA can be made accessible to cooperating CAs by issuing explicit cooperation operations (as mentioned in Sect. 2.4).

Imagine a group of persons is to write a paper. For that purpose, CGE can be used to create a CA all group members can be associated with. An ‘empty’ text is created, may be named “paper”, and its completion is considered being the task of the newly created CA. Now users start working on the document by applying CGE functions. Usually they start with agreeing in a certain structure of the text and with delegating parts to members of the group. Initially the text consists of a single segment. Working in CGE means manipulating a segment being part of the text associated with the team. We distinguish *explicit segments* from *implicit segments*. Explicit segments are created for cooperation purposes; this matter will be discussed in the following subsection. An implicit segment is the part of an explicit segment currently accessed by the CGE modify-function. Thus, implicit segments cope with synchronizing actions of users closely cooperating with each other, i. e. (so far), belonging to the same CA. Let us assume for the remainder of this subsection to only consider actions initiated by members of the same CA. Reading segments just means transferring the data to the CGE frontend and committing the read transaction immediately. Thus, multiple users may read the same segment. During text modifications the following rules are observed:

- The part of the segment to be manipulated must be explicitly or implicitly marked; it is important to mark exactly that excerpt which is semantically concerned by the intended modification; this can usually only be decided by the user who initiates the modification function; marking explicitly is done by mouse, marking implicitly means that a pre-setting leads to expand the cursor position to the corresponding word, sentence, paragraph or user-defined excerpt, e.g. the text between two headers of a certain format.
- The frontend tries to create an implicit segment containing the marked text. If the frontend realizes that the user has marked within an out-of-date version (this occurs, if the user's frontend content is not in accordance with the corresponding content of the database w.r.t. to that segment) his frontend representation is brought up-to-date and all parts currently under modification by other users are correspondingly visualized (i. e. the user can get the information, which cooperating user is currently working on that part). The system keeps read locks on the (remaining) segment-data for a certain duration within which the user has to decide whether or not the intended modification is still reasonable.
- The user may now choose free parts of the current segment state for manipulation; he may also choose from parts currently in work by a cooperating user; this leads the frontend to simultaneously visualize the actions issued by the cooperating user (provided the latter permits 'observation'); in this way, the user can see, what the cooperating user is doing, but can only manipulate that part of the text he write-locked.

Now we can summarize the principles of manipulating text within a group of closely cooperating users, e. g. the users associated with the same CA:

- *Short read transactions*: a segment-reading transaction is committed as soon as the frontend received the content.
- *Modifying semantic units*: the processing structures rely on the assumption that the user is able to mark exactly the part of the text, which is semantically concerned by the intended modification.
- *Demand-updating of frontend representation*: since the frontend contents can become outdated with the release of manipulations done by cooperating users, the topic version is reloaded as soon as required.
- *'Looking a cooperating user over the shoulder'*: A user can watch the modifications of a cooperating user, but can only change text he has currently write-locked.

Loose Cooperation

An explicit segment is created by issuing the corresponding CGE function, extracting the (marked) part of the text and establishing a sub-segment of the one currently opened by the user. For example, creating a sub-CA usually requires to extract a sub-segment and to delegate it to the newly created CA. Thus, the new CA is aimed at fulfilling the task of completing the received segment and passing it back as enforced by the protocol underlying the delegation relationship type. In this way, a hierarchy of CAs can be dynamically established corresponding to text decomposition and team structure (members of the team need to be associated with certain CAs in order to fulfill the task of completing the segment). Fig. 2 visualizes the creation of a sub-CA. Suppose, a top-

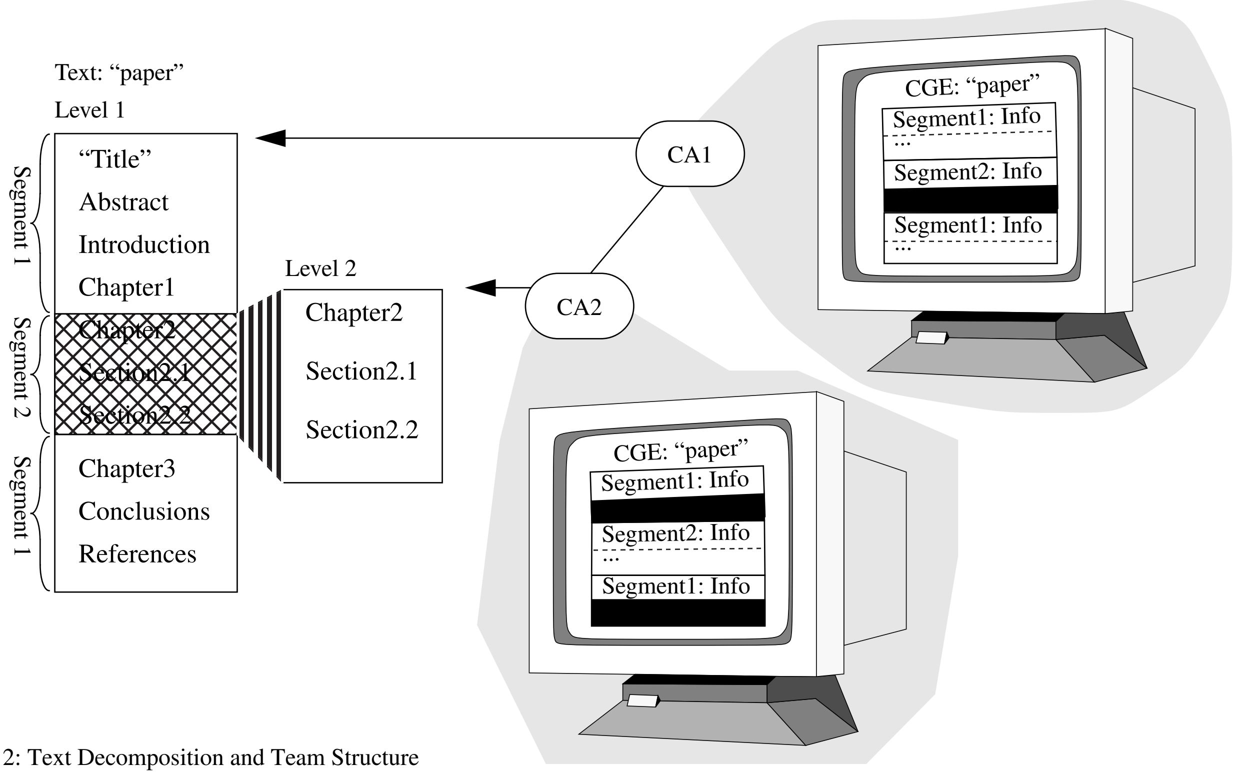


Fig. 2: Text Decomposition and Team Structure

level CA (CA1) has been created to write the text “paper”. After having determined the structure of the paper (note that the text structure is not a kind of meta-data), chapter2 is supposed to be delegated. For that purpose, CA2 is created, one or more team members are associated with the newly created CA, a corresponding explicit segment chapter2 is created and delegated to CA2. This constellation is depicted at the left-hand side of Fig. 2. The right-hand side of Fig. 2 illustrates, what members of the different CAs may see after having read the document. The (explicit) segment structure is shown on the screen, an info-bar gives information about the corresponding segment, but contents, which are not in the scope of the CA, cannot be accessed (black text bars in Fig. 2). Now users can work on the text (segments) which is (are) associated with their respective CA in the way described in Sect. 3.1. After having finished chapter2, CA2 is expected to pass back the corresponding text to be integrated again into the overall document.

By use of the text-segmentation functions and the functions allowing to manipulate sub-CAs, a hierarchy of concurrently active tasks can be dynamically established. So far, we reported on the cooperation possibilities between users of the same CA. In the following, we will report on the facilities enabling cooperation between CAs, i. e. facilities for loose cooperation.

Provision of State Information

Since users associated with a particular CA may not access data assigned to another CA, they are at any time allowed to acquire (meta-) information about the current state of the overall process. This contains information about the current CA hierarchy, the users associated with CAs, the tasks of the CAs (responsibility for particular text parts), and the *state* of a text segment in work by a particular CA. The state gives some idea of how far the CA is away from finishing the segment. Each segment delegated to a CA gets the default states *start*, *preliminary* and *finished*. Users may refine the state information and attach a textual description/ explanation to the state identifier. For example, a state ‘almost finished’ may stand for ‘text must only be supplemented by some further literature references’. During work users should set text states to show others whether or not the text fulfills qualitative prerequisites for a cooperative exchange.

Communication / Notification

Thus, CGE provides functions to get the information which text states are defined for a particular segment and which state is currently set for that segment. Now, if the segment is in a state promising any benefit to cooperate on, a user of another CA may send a request to the owning CA, applying for getting access (in read mode or even in write mode) to that segment. If, on the other hand, the segment is not in the wanted state yet, CGE may be instructed to send a notification as soon as the particular text state is reached.

Cooperation (Granting Access)

After having received a request, a user has to decide on whether or not access can be granted, and, if so, which mode (read or write) is supposed to be provided. CGE allows to grant (and revoke) certain access on own segments to certain cooperating CAs.

Thus, due to cooperation facilities, not only the users associated with the CA owning a segment may access the text, but also users of a cooperating CA, which got corresponding access rights. If write access has been granted, the users of the involved CAs cooperate with each other (w.r.t. the corresponding text) as described in Sect. 3.1 (Close Coop.).

Further Features

Due to space restrictions we can only report important features of CGE. Besides the mentioned cooperation possibilities, CGE provides a rich spectrum of functions for

- manipulating the CA hierarchy, user associations (team structure) as well as text structures;
- manipulating text;
- notifications, direct communications, and messages;
- annotating text;
- logging and providing history information (which users read/ modified which segments at which time).

CONCURRENCY CONTROL AND CONSISTENCY MAINTENANCE

After having introduced major CGE functionality, we now want to report on the concurrency-control aspect.

Mapping CGE Functions to Database Transactions

Obviously, user sessions cannot directly be mapped to transactions, since the DBMS provides ACID transactions, which would completely isolate users from each other. Therefore, we decided to chose a smaller operational granule and mapped single CGE functions, such as reading a segment or modifying a text fragment to transactions, respectively. This ensures basic consistency, but introduced two new problems:

- editor and database representation of a segment may differ, due to concurrent manipulations;
- locks must be acquired on flexible text granules, in order to lock as few data as necessary and, thereby, enable as much concurrency as possible.

In the following, we will outline, how these problems of data replication and flexible locking granules have been solved in CGE.

editor block:

segment-no	segment-level	version-no	text

database block:

segment-no	segment-level	version-no	lock	text

Fig. 3: Blocks

Modeling Editor Blocks and Database Blocks

Blocks are portions of text (implicit or explicit segments) and serve as data manipulation units. There are editor blocks shown

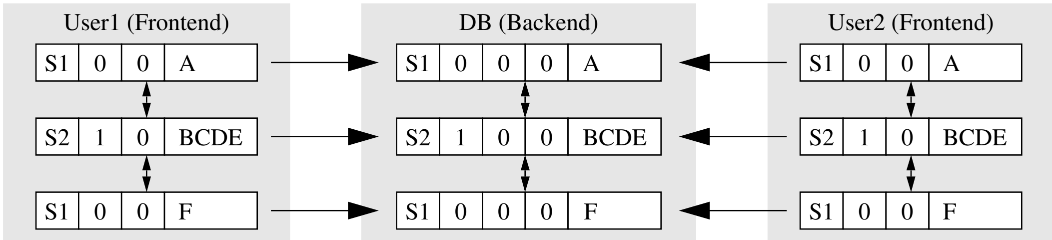


Fig. 4: Initial State of a Sample Scenario

to the user and manipulated by the CGE frontend as well as database blocks. These blocks embody the structures illustrated in Fig. 3:

segment-no: number of an explicit segment; note that creating a sub-segment means pulling out a text part; this implies that the original segment keeps the text parts, which were before and after the cut out part;

segment-level: refers to the level of the explicit segment w.r.t. the overall text structure;

version-no: to each block (also blocks representing implicit segments) a version number is assigned, which is incremented by each successful write operation;

lock: indicates, whether or not the corresponding block is currently locked by a modify operation; actually the lock entry is given as the pair (CA-Id, User-Id) in order to provide information to other users if necessary; due to simplicity we will consider it in the following as a flag; the lock entry is only important for database blocks, since it must be decided, whether or not the contents may be delivered to a frontend after a request;

text: refers to the contents of a block containing text and format information.

Concurrency Control

In the following, we want to outline the concurrency-control mechanism exploited in CGE by considering a sample scenario illustrated in Fig. 4. The database contains a text ABCDEF, where the capital letters stand for arbitrary contents. Thus, explicit segment S1 refers to the overall text. A sub-segment S2 has been cut out containing text BCDE. This leads to segmentation of the text into database blocks depicted in the middle of Fig. 4. Let us assume, there are two users performing CGE sessions editing the text. Both users issued the CGE read function on the overall text and got the complete contents, so that both frontends manage editor blocks, each equivalent to a database block (see Fig. 4).

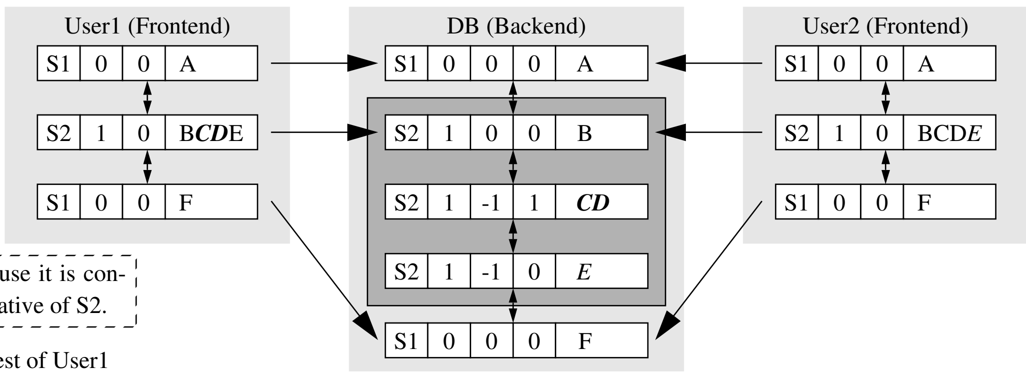
Let us now assume that user1 starts a modification on text CD. Since there are no conflicts, the modification can be allowed. As a consequence, S2 is implicitly segmented (see database blocks depicted in Fig. 5). Implicit segmentation can be recognized by equal segment numbers and negative version numbers. The latter indicate that the considered block does not correspond to an explicit segment and, therefore, has only been created for synchronization purposes.

Next, user2 intends to modify the text part DE. The system now has to perform the following steps.

1. *Version-Check*: It has to be checked, whether or not the user's frontend representation is up-to-date. This is found out by comparing the version numbers of the frontend representation and of the corresponding database segment. Since these are equal in the example, it can be concluded that the user sees the current text version and that the frontend representation does not need to be brought up-to-date.

2. *Lock-Compatibility-Check*: Checking Compatibility requires finding the concerned text part (possibly an implicit sub-segment) at the database side, what has to be done by text comparisons. In this way, the system determines the last two implicit sub-segments (CD, E) of explicit segment S2 (BCDE) to be checked. It is detected that part D is currently under modification by user1 (see lock entry).

3. *Conflict-Resolution*: Since a conflict has been detected in step 2, the system provides the information to user2 that part D is currently in use by user1 and offers him to manipulate the last implicit sub-segment of S2 (E). During the decision period the corresponding segments are locked so that no third user can invalidate the offer. User2 now has to decide whether or not the intended modifications are still reasonable. He has the possibilities to reject the offer or to accept the restricted modification area (last implicit sub-segment with contents E). Furthermore, he can request a window showing him the current manipulations done



Note: Despite implicit segmentation of S2, the first S2-database-block (B) does not get a negative version number, because it is considered to be the representative of S2.

Fig. 5: Situation after Request of User1

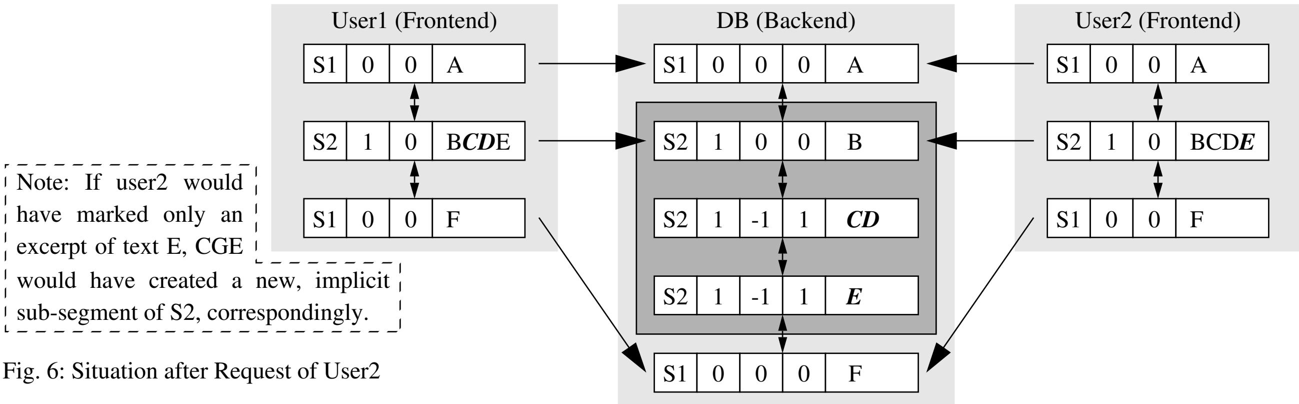


Fig. 6: Situation after Request of User2

by user1 on the second implicit sub-segment (referring to text CD). Note that conflict resolution does not apply blocking, but asks the requesting user to do a dynamic (manual) resolution [21].

4. *Managing Modification Area*: Supposing, user2 decides to modify part E, he is enabled to do so, i. e., the constellation shown in Fig. 6 occurs. Note that this step does not require data transfer between backend and frontend, since contents are already equivalent (at the latest since step 1).

Next, user1 finishes his modifications. The system releases the lock on the second implicit sub-segment of S2 (CD) and tries to merge it with (unlocked) neighbored, implicit sub-segments. In our example, this leads to a merge of the first (B) and the second (CD) sub-segment. After user2 also finishes his modifications, the situation illustrated in Fig. 7 is given, in which all implicit sub-segments mentioned above are again integrated into the explicit sub-segment S2 and the text CDE has been modified.

The fonts used in Fig. 7 illustrate, which (parts of) editor blocks are up-to-date and which are not. Suppose, user1 next wants to modify part E. The system then finds out by comparing the version numbers that the frontend representation is outdated. Thus, the frontend representation is brought up-to-date and the user must decide again, whether or not the intended modification is still reasonable.

The example shows the basic principles used in CGE to control processing:

- the lock granule is flexible (implicit segments) and is determined by the intended modification;
- implicit blocking of user activities does not occur;
- conflict resolution can be done early (foresighted) and in a flexible manner, since besides automatic reactions also users can be involved to resolve conflicts dynamically (manually);
- modifications are propagated to other frontends in a lazy fashion, i. e. not before they are meaningful and, therefore, important for the corresponding user.

In this way, an adequate harmony of cooperation support and enforcement of data consistency is achieved.

Implementation

CGE has been implemented in C++ on SUN workstations running the operating system UNIX. Tk/TcL has been used to provide the frontend (state information is visualized by using colors), data is managed by the object-oriented database management system ObjectStore.

CONCLUSIONS

CGE exploits a similar document decomposition as used in the DUPLEX approach [18]. But in addition to explicit segmentation, implicit segmentation is introduced to support the technical concurrency-control protocol. Cooperation facilities, which are installed at a higher abstractional layer as the concurrency-control mechanisms, deal with user operations and user objects (explicit text segments), and, additionally, are explicit and dynamic. Thus, cooperation control fits into the mental model of users. In the same way, concurrency control does not lead to bothering effects at the interface, although basic (text-manipulating) operations are isolated against each other. The CGE concurrency-control mechanism fulfills the requirements mentioned at the beginning: no implicit blocking, early conflict detection, flexible resolution by involving users into the conflict management process as far as concurrency control on user objects is concerned. Thus, the used mechanism enforces isolation only in situations in which users want to be isolated (and to work undisturbed) and en-

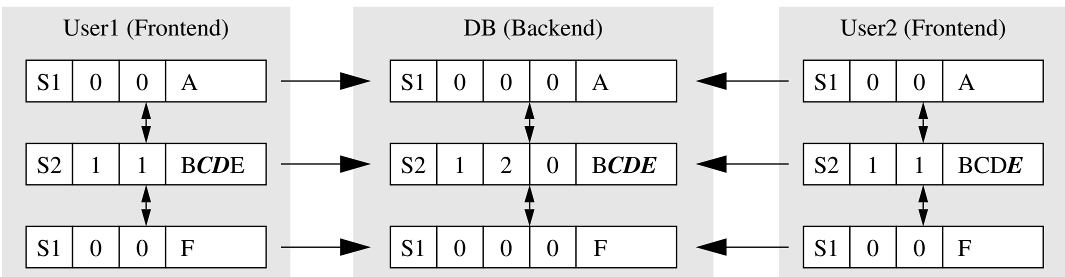


Fig. 7: Resulting State

ables cooperation in situations they want to exchange preliminary information.

First experience in using CGE confirms that the used technical protocols are not contrary to natural processing but agree with social protocols. Nevertheless, in the future additional case studies will be performed. Furthermore, we want to deal with problems occurring with decentralizing data management and using heterogeneous data management facilities. Up to now CGE can just be used within a LAN (local area network) environment. Extending CGE to be usable in large-scale networks will also mean to cope with problems of communication delays, node crashes and data-transfer failures.

Acknowledgments

The author would like to thank T. Härder and H.P. Steiert for their helpful comments on an earlier version of this paper and T. Krech for managing the CGE implementation.

Literature

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

2 Ellis, C.A.; Gibbs, S.J.: Concurrency Control in Groupware Systems, in: Procs. of ACM SIGMOD Int. Conf. on Management of Data, Portland, OR, 1989, pp. 399-407.

3 Ellis, C.A.; Gibbs, S.J.; Rein, G.L.: Design and use of a group editor; Engineering for Human Computer Interaction; Cockton, G., Ed., North Holland, Amsterdam, 1990, pp 13-25.

4 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.

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

6 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, Chapel Hill, NC, 1994, pp. 207-218.

7 Greif, I., Seliger, R., Weihl, W.: A case study of CES: A distributed collaborative editing system implemented with Argus, IEEE Transactions on Software Engineering, 18(9), pp. 827-839.

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

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

10 Knister, M.J., Prakash, A.: DistEdit: A distributed toolkit for supporting multiple group editors, in: Procs. ACM Conference on Computer-Supported Cooperative Work, Los Angeles, CA, 1990.

11 Koch, M.: Design Issues and Model for a distributed Multi-User Editor, Computer Supported Cooperative Work, International Journal, 5(1), 1996.

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

13 Lu, I.M., Mantei, M.M.: Idea Management in a Shared Drawing Tool, in: Procs. 2nd Europ. Conf. on Computer-Supported Cooperative Work, Amsterdam, The Netherlands, 1991, pp. 97-112.

14 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.

15 Neuwirth, C.M., Chandhok, Kaufer, S.D., Erion, P., R., Morris, J.H.: Flexible diff-ing in a collabortive writing system, in: Procs. ACM 1992 Conference on Computer Supported Cooperative Work, Toronto, Canada, 1992, pp. 147-154.

16 Neuwirth, C.M., Kaufer, S.D., Chandhok, R., Morris, J.H.: Computer Support for Distributed Collaborative Writing: Defining Parameters of Interaction, in: Procs. ACM 1994 Conference on Computer Supported Cooperative Work, Chapel Hill, NC, 1994, pp. 145-152.

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

18 Pacull, F., Sandoz, A., Schiper, A.: Duplex: A Dsitributed Collaborative Editing Environment in Large Scale, in: Procs. ACM 1994 Conference on Computer Supported Cooperative Work, Chapel Hill, NC, 1994, pp. 165-174.

19 Ramamritham, K., Chrysanthis, P.K.: Executive Briefing - Advances in Concurrency Control and Transaction Processing, IEEE Computer Society, Los Alamitos, CA, 1996.

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

21 Ritter, N., Mitschang, B., Härder, T.: Conflict Management in CONCORD, 6th. Int. Conference on Data and Knowledge Bases for Manufacturing and Engineering, Tempe, Arizona, Oct. 1996, pp. 81-100.

22 Rusinkiewicz, M., Klas, W., Tesch, T., Waesch, J., Muth, P.: Towards a Cooperative Transaction Model - The Cooperative Activity Model, in: Procs. 21. Int. Conf. on Very Large Data Bases (VLDB), Zurich, Switzerland, Sept., 1995, pp. 194-205.

23 Sasse, M.A., Handley, M.J., Chuang, S.C.: Support for Collaborative Authoring via Email: The MESSIE Environment, in: Procs. 3rd European Conference on Computer-Supported Cooperative Work, Milan, Italy, 1993, pp. 249-264.