Database and Transaction Model for Dynamic and Cooperative Workflows

Waldemar Wieczerzycki

Department of Information Technology, University of Economics at Poznań, Poland

In the paper, a flexible persistent environment for workflow system applications is proposed. It enables both efficient business process modeling and execution. A special emphasis is put on the dynamic evolution of process description over a given time, on one hand, and on the cooperative execution of activities embedded in processes which correspond to transactions, on the other.

In case of activity evolution, the proposed approach is based on the notion of the database configuration, which comprises versions of objects included in a particular set. The paper shows that a particular versioning technique may improve and simplify process modeling.

In case of activity execution, new ideas concerning transaction management are proposed, which aim to efficiently and widely support the cooperation between users assigned to the same tasks, who intend to achieve common goals in a collaborative way.

Keywords: Workflow, CSCW, Transaction Management, Object-oriented databases

1. Introduction

Most of the problems faced by enterprises concern internal business procedures that are neither well defined nor particularly efficient (Hales 1993, Lavery 1992, Medina-Mora et al. 1993). Workflow management is a computer-based, potential solution to these problems. It is a system for managing a series of tasks (actions) defined for one or more procedures (Aiello et al. 1984, Brierley 1993, Bullinger et al. 1993, Hendley 1992, Jones et al. 1993). The system ensures that the tasks are passed among the appropriate participants in the correct sequence and completed within set times (default actions being taken where necessary). Participants may be people (actors) or other systems. People normally interact through workstations while other systems may be either on the same computer as the system considered, or on another accessible over a communication network.

Many tasks performed by enterprises can be effectively automated using current business-oriented workflow management products (Sheth 1994, Kling 1991, Hennessy et al. 1989) which typically support imaging, document processing and routing. Some enterprises perform tasks that must be modeled in a client-server style, using a transaction mechanism. These can be supported by workflow systems built on top of the distributed databases (Georgakopoulos 1994, Huhns et al. 1994, Shuster et al. 1994). Finally, workflows are often pro-active and embed long-living processes. Process agents, i.e. members of the organization structures of the enterprise, are in charge of executing processes and process steps. Workflow management systems have to associate appropriate process agents to process pro-actively. That is why they are built over active databases (Bussler et al. 1994).

The emerging computing paradigm of workflow management is beginning to influence not only business-oriented applications, but increasingly those in the scientific sector, too. Scientific applications present particular problems of workflow management. Not only do the workflows change frequently, but their refinement may only emerge as a result of experimentation as the production/workflow process is followed (McClatchey et al. 1997). Scientific workflows may be specified and executed in an ad-hoc manner, may be aborted during execution and often differ in nature from their definition at the outset of the scientific experiment. New definitions of activities can be added to the workflow system and existing definitions can be amended or

deleted as the workflows evolve. This is usually called dynamic modification of workflows (Vossen et al. 1996). For these reasons and others commercial workflow management systems appear to be inadequate for the purposes of managing scientific workflow applications. In particular, these products are rather restricted in coping with the versioning problems which arise when workflows change on a frequent basis, i.e. when workflow models evolve dynamically, possibly towards many different directions (Benford 1991, Smith et al. 1989).

Besides the support for dynamic evolution, workflows require also support for cooperative work of people collectively assigned to perform particular activities (tasks), e.g. writing a co-authored report, collaborative preparation of a business contract, etc. Since many workflow management systems are transactional, traditional transaction models become insufficient. New mechanisms supporting efficient cooperation of long-duration transactions working in both shared and exclusive environments must be provided. In particular, conflicts between transactions of this type should be avoided whenever possible. If, however, conflicts do occur, they must be resolved in such a way that transactions are not suspended or rolled-back, but that their execution may be immediately continued.

In the paper workflow systems supported by object-oriented databases are considered. The first main contribution of this paper is a particular approach to business process modeling in workflow systems, which provides new concepts and operations that are necessary for dynamic evolution of workflows. The approach is based on the notion of the database configuration, which comprises versions of objects included in a particular set. The paper shows that a particular versioning technique may improve and simplify both process modeling and execution.

The second main contribution of the paper consists in a new approach to transaction management, which aims to efficiently and widely support the cooperation between users assigned to the same tasks who intend to achieve common goals in a collaborative way. Briefly, it can be fulfilled by assigning the entire group of collaborating users to the same transaction, still preserving, however, the identity of individuals. The proposed transaction model is inspired by the natural perception, that a team of intensively cooperating users can be considered as a single virtual user who has more than one brain trying to achieve the assumed goal, and more than two hands operating on a keyboard.

The paper is organized in the following way: in Section 2 basic concepts are given and a process modeling technique oriented for dynamic evolution of worklows is proposed; in Section 3 a new transaction model supporting cooperation between users, as well as particular transaction management mechanisms are discussed; in Section 4 brief experimental evaluation of the proposed approach is described; finally, in Section 5 concluding remarks are given.

2. Process Models and Instances

2.1. Basic Concepts

Dynamic workflow evolution may be seen in different ways. First, it may be seen in terms of improving the process description, elementary task allocation and scheduling, resource utilization, etc., in order to increase the quality of services offered and, as a consequence, the total income of the enterprise. The improved process description typically replaces the previous one, next it is validated and finally, depending on the results achieved, accepted or rejected. In case of unsatisfactory results, process roll-back must be done in order to restore its previous description. It means that old descriptions must be somehow kept in the system. Even in the case of satisfactory results of process validation, an older description may be useful in the future, due to the cyclic nature of process or returning requirements.

On the other hand, process evolution relates to the creative nature of processes. Most of them usually create and develop very complex objects which are the expected outputs (artifacts) of processes. The progress is reached step by step, through the creation of improved versions of the object being developed. Similarly to other approaches, our model of an enterprise is composed of two parts: a model of organizational environments and a model of processes, which are strictly related to each other and mutually dependent (Hawryszkiewicz 1994). *Environments* model the support structures for groups

of employees. Environments can be of a prolonged duration and support a variety of processes. They can also define the social context for cooperation. *Environments* can include other environments, as well as artifacts, roles and actors. *Artifacts* represent the information base of the enterprise. These can include files as well as artifacts such as reports, designs, and so on. *Roles* are abstract entities that represent system decisions. For example, purchasing a part requires a purchase requisition to be made and approved. The requisition and approval decisions are modeled as two separate roles. An actor is assigned to each role to make the decision. Actors are often positions that can be derived using organizational rules. Roles are not permanently associated with positions but are dynamically assigned using organizational rules.

Each *process* corresponds to a subset of actions performed by the enterprise to achieve one of the goals it has been created for. Each process is within an environment and models interactions between a subset of objects in this environment. Processes are further decomposed into activities. Activities correspond to different stages of process execution. They are partially ordered according to ways the process is executed in real-life. In order to be initiated, some activities require particular artifacts as an input, which may be taken directly from corresponding environments or produced as outputs by other activities. Moreover, some activities are triggered by so called events, which may be classified into external events (e.g. a client's request), internal events (an employee's decision), and time events (a change of the year). In our approach, activities are only units of process decomposition. In general, however, they may be further sub-divided into so called *tasks*, which correspond to elementary, well-defined actions in the scope of a given activity.

2.2. Dynamic Process Evolution

In this section we will show how basic concepts presented in the previous section can be modeled in a multiversion database supporting workflow management applications, and how they can evolve over time, due to dynamic modifications of their models.

Every process is represented in the database by its precise description, which is called the *process model*. This description contains the structure and behavior of all objects that may potentially be used during process execution, or produced as a result of process execution, e.g. artifacts, roles. It also contains the exact specifications of all activities embedded in a process, their scheduling and activation rules, and their inputs and outputs. In database terms the descriptions of all processes modeled constitute the database scheme.

Processes are not only modeled in the database to reflect static features of a real-world enterprise, but they are also executed in order to show how real-world employees perform their official duties, and how the enterprise as a whole is trying to achieve its strategic aims. Every process execution is represented in a database by a so called *process instance*, which is created and evolves according to the information included in a corresponding process model. In general, every process (client acquisition, advertising, etc.) may be performed simultaneously and independently by different actors. Thus, a single process which is always described by a single process model, may have an arbitrary number of instances which live in the database asynchronously. Process instances will be more deeply considered in Section 2.3.

Every process model evolves during the database life-time in the same way corresponding execution rules are improved in the enterprise, in order to maximize its efficiency, profits, quality of services, etc. This process has a progressive nature, which means that it is performed step by step through the creation of improved versions of process models. New versions of process models, however, do not just replace older versions — they are kept parallel to them. Preserving historic process model versions makes it possible to perform roll-backs in process model evolution, e.g. due to unsatisfactory results obtained. On the other hand, sometimes there is an evident need to return to previous versions of process models due to a requirements change or other factors which often have a cyclic nature, e.g. a process may be executed always in the same particular way on every December and differently during the rest of a year.

There is also another aspect of process model evolution. A single process may be sometimes performed in slightly different ways in the same time period, depending on triggering events, particular conditions, temporary results obtained, etc. In this case, instead of a single, most recent version of a process model, we deal with many *variants* of the same process model. Contrarily to historic versions which reflect the progressive nature of processes, variants reflect their alternative nature. Another difference is that historic versions are normally frozen and labeled by particular time stamps, being sort of snap-shots in the enterprise history, while variants may be freely updated.

Most of the updates addressed to a process model consist of changing their state by accessing objects belonging to them. These updates are called *non-versioning* updates, due to the fact that neither a new version of the whole activity model is derived, nor are new versions of objects composing it created. There are also updates, called versioning updates, which derive new versions of a process model on the basis of a parent one which is preserved in the database in its original state.

From the user's point of view, a database supporting workflow system is composed of a database background and a set of multiversion process models. The *database background* models an enterprise without processes. It contains all environments which may be used to define a new process, which in turn are composed of objects like actors, resources available, standard roles that are well defined, however not performed yet, etc. In other words, the database background contains all objects which may be useful to execute future processes and can be foreseen since the beginning of enterprise existence.

Every process model describes exactly one enterprise process. Initially, i.e. after the so called process initialization, it is composed of logical copies of some objects contained in the database background. Afterwards, due to the detailed process definition, it is dynamically extended by new objects local to the process being defined, reflecting the specificity of the process. In particular, it is extended by a specification of all the activities that may be performed in frames of the process. Finally, it is also extended by the semantic relationships among the objects mentioned above which order activities, assign them particular roles, point artifacts that are exchanged between activities or used internally by an activity, bind external and internal events to actions which they trigger, etc.

A database example in a way perceived by the user is illustrated in Fig. 1. On the basis of the database background, four processes have been initialized. Next, the definition of every process has been refined by the creation of its new versions. Every process is represented in Fig. 1 by a stack of soft boxes; each box corresponds to a single process model version. Initial versions, filled in the same way, are put at the bottom of every stack. Also, the most recently released process model versions, which are put at the top of stacks, are especially distinguished. The model of process *P4* is available in six versions: four frozen historic versions and two current versions, which represent process variants.

All processes are derived (initialized) from the same database background by selecting subsets of its objects. These subsets need not be disjointed i.e. some objects stored in the database background may be shared between activities. Updates in the database background are allowed and are automatically propagated to those processes which share objects affected. On the contrary, newly created objects in the background are not included in the processes already initialized. They may be used in subsequent initializations of new processes. Updates in processes are local, which means they do not influence the background and other processes.

Current versions of process models are grouped together and are called the *database framework*. The database framework represents current view of the enterprise, i.e. all processes that are currently executed or may be executed in the near future. Typically processes are not quite isolated from each other; similarly to activities composing a process, they may also be partially ordered or one may observe an artifact flow between them. The notion of the database framework supports these relationships which are outside the processes but inside the framework.

Because of different levels of abstraction, the user's view of the database is different from that of the database management system (*DBMS*). In the latter case, one must discuss what is the

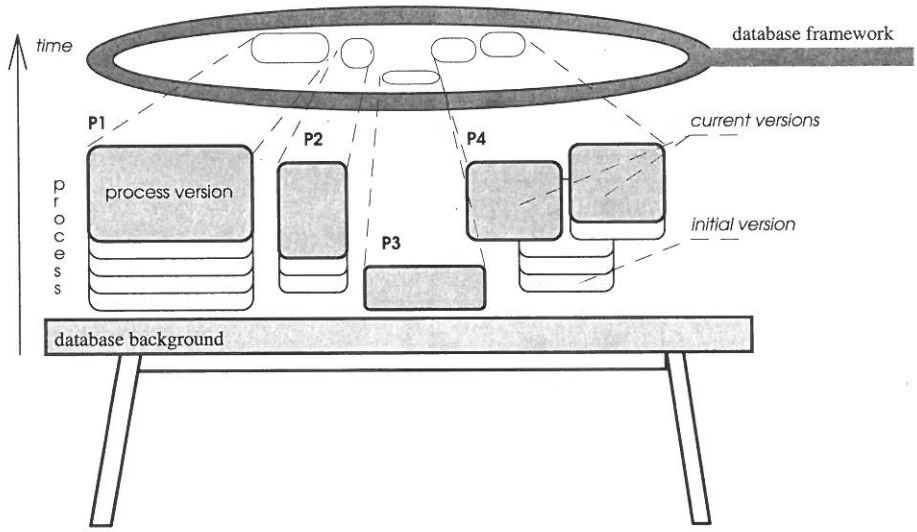

Fig. 1. User's View of the Database

granule of database versioning and what is the unit of the database consistency.

At the *DBMS* level the database is viewed as a tree of so called *database configurations* (cf. Fig. 2). Every configuration is composed of single versions of objects belonging to a subset of objects stored in the database, and corresponds to a single process model version. The root configuration corresponds to the database background. Every object is always addressed in the context of a previously chosen configuration. Non-versioning updates performed on it are local to this configuration — they do not affect other versions of the same object belonging to other configurations. It also concerns updates performed on object versions that are physically shared between configurations; in this case a new object version is created which is local to the configuration addressed, while the old object version remains unchanged and is still available in other configurations. As a consequence, database configurations are mutually independent and they may evolve without imposing any constraints on the evolution of other configurations. The only exception concerns the root configuration (the configuration modeling the database background); updates performed on shared objects belonging to it are propagated to respective child configurations.

The situation is different in the case of versioning updates. Every update of this type is preceded by the automatic derivation of a new (child) configuration which is initially identical to its parent, i.e. which is a logical copy of its parent. Afterwards, the update is performed in the child configuration without affecting objects in the parent configuration. In other words, after the versioning update, a new configuration appears in the system which shares all object versions with its parent, except the one affected by the update operation (cf. Fig. 2).

From the afore-mentioned discussion it follows that a set of database objects is the unit of versioning — every version of this set becomes a single database configuration. It is not possible to derive an object version out of the scope of a particular configuration. Notice that in two extreme cases a versionable set may be composed of all objects stored in the database or of a single object.

2.3. Dynamic Process Instantiation

One aspect is process modeling in order to precisely represent it in the database, and another aspect is process execution according to a previously accepted process model. In this section we will focus on the so called process instantiation which starts the execution of activities embedded in the process according to patterns included in the process model. The emphasis will be put on additional modeling

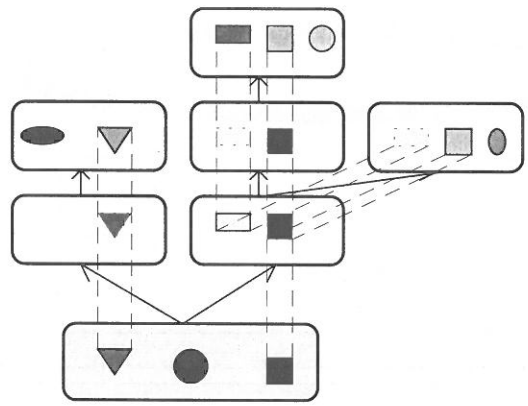

Fig. 2. DBMS View of the Database

mechanisms rather than transaction management mechanisms, which we discuss in Section 3.

As mentioned before, every process may be composed of an arbitrary number of elementary activities which are partially ordered. It is also assumed that, in general, processes are interactive and require intensive information exchange among users, who are modeled in the system as actors. One may distinguish three types of processes represented in the database. The first type contains read-only processes that read objects stored in the database, but do not update them and do not create new objects. Of course, processes of this type may produce new information, e.g. reports, letters, management directives, etc., displayed or printed to the users. The second type contains processes that may create new persistent database objects that are local to them, i.e. that exist only during process execution and play the role of artifacts exchanged between activities in the scope of the same process. The third, most general type, contains processes that may both update existing database objects and create new persistent objects that are retained in the database after process execution.

The main aim of the proposed approach is to avoid conflicts between processes when they are executed, and between processes and database operations described in the previous section, like process initialization and definition. Most conflicts arise when different processes try to access the same objects. Conflicts do not occur between processes of the type one case, they may occur between processes of the type two case, and they are very possible between processes of the type three case. The most straightforward solution to avoid conflicts is to isolate processes by addressing them to different database configurations. It is relatively easy in the case of different processes, while not so obvious in the case of processes being instances of the same process model, which are executed asynchronously (cf. Section 2.2).

To avoid conflicts between instances of the same process model, a new database configuration is automatically derived whenever a process is instantiated, directly from the configuration which represents a corresponding process model. This new configuration becomes an exclusive scope in which a newly instantiated process is executed. Initially, it comprises logical copies of all objects included in the parent configuration, i.e. it contains all patterns included in the process model. Afterwards, it may evolve due to the changes implied by process execution.

The process instantiation is illustrated in Fig. 3. Process P_1 is represented by a single model version. It has three instances: $i1$, $i2$ and $i3$, which are executed separately. Process instance $i1$ is currently performing activity $a1$, instance $i2$ is performing in parallel activities — $a2$ and $a4$, while process instance $i3$ has been finished. The last one has created three new objects, represented by circles, which are available only in the database configuration representing $i3$. Notice, also, that parallelly with the execution of those three process instances it is possible to refine the corresponding process model which is included in the parent configuration logically independent from its children.

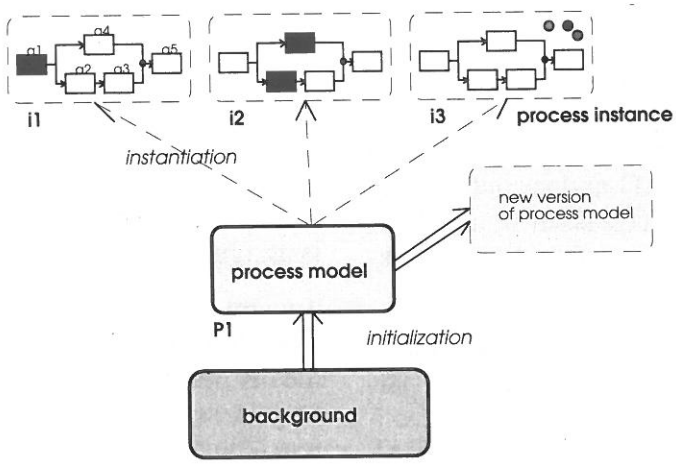

Fig. 3. Process Instantiation

Process initialization consists of the creation of a new database configuration on the basis of the database background. Process definition consists of refining a process model by adding to it local objects specific to the process and semantic relationships between them. Activity instantiation first creates a new database configuration and exclusively dedicates it to a single process execution and then starts the execution of respective activities embedded in the process.

3. Process Execution

3.1. Survey of Transaction Models

A transaction is an elementary unit of interaction between the user and the *DBMS*. There are many advanced transaction models proposed in the literature (Elmagarmid 1992). Two early non-traditional models are: nested transactions and Sagas. *Nested transactions* (Moss 1985) support modularity, failure handling and intra-transaction parallelism. They are very important in the development of other, more advanced models, e.g. *ACTA* model (Chrysanthis et al. 1990), because they introduce the idea of structuring a transaction into a tree (or hierarchy) of subtransactions and non-vital subtransactions. *Sagas* (Garcia-Molina 1987), based on the compensating transactions, consist of a set of subtransactions. They relax the property of isolation by allowing a saga to reveal its partial results to other transactions before it is complete. Sagas are useful only when subtransactions are relatively independent (because of consistency problems) and each subtransaction can be successfully compensated.

There are some advanced transaction models which are addressed to distributed databases and multidatabases. *DOM transactions* (Buchmann et al. 1992) extend the concept of nested transactions proposing, so called, closed nested subtransactions, open nested subtransactions, and combinations of the two. *Flex transaction* model (Elmagarmid et al. 1990) allows the user to specify a set of functionally equivalent subtransactions, each of which when completed will accomplish a particular task. This model also allows the specification of dependencies on the subtransactions. *Polytransactions* (Rusinkiewicz et al. 1991) facilitate the support of interdependent data in multidatabase environments. Interdependent data is defined to be two or more data items stored in different databases that are related to each other through an integrity constraint.

Finally, there are transaction models supporting cooperation between transactions. The most general approach proposes the *cooperative transaction hierarchy* (Nodine et al. 1984) which allows to associate transactions encompassed by a transaction group with individual designers. The notion of correctness defined by serializability is substituted by the notion of user-defined serializability. Because isolation between transactions is not required, the transaction hierarchies allow close cooperation between transactions and also help to alleviate the problems caused by long-lived transactions.

Other transaction models from this group are not so general, since they are addressed to particular application domains. *Cooperative SEE transactions* (Hill et al. 1992) were developed for software engineering environments. *ConTract* model (Reuter 1989) is mainly addressed to office automation, CAD and manufacturing control. *S-transactions* (Eliassen et al. 1988) support cooperation in the international banking system.

Taking into account the needs of cooperative workflows in which some activities are performed by groups of collaborating users, cooperative transaction hierarchies (Nodine et al. 1984) are very promising, since transactions form the same group are not isolated mutually and can correspond to different, though somehow related tasks. Notice, that this approach, similarly to all other approaches mentioned above, uses a tree-structured transaction model.

An attempt to apply hierarchical transactions to databases has some disadvantages. Contrarily to flat transactions, hierarchical transactions require sophisticated transaction management methods and, as a consequence, additional system overhead which reduces its performance. Moreover, hierarchical transactions are still not sufficient, considering expectations of cooperating users, since in many situations the transaction correctness criterion restricts wide cooperation. Finally, they are not so reliable as flat transactions, since in practice commercial databases use the latter ones.

In further subsections of this chapter we propose a solution of problems mentioned above by the use of flat transactions, in which, in comparison to classical ACID transactions (Gray 1978), the isolation property is relaxed.

3.2. Multi-user transactions

A *multi-user transaction* is a flat, totally ordered set of database operations performed by a group of users (a team) assigned to the same activity, which is atomic, consistent and durable. In other words, a multi-user transaction is the only unit of communication between a virtual user representing members of a single team, and the database management system.

Formally, a multi-user transaction is defined as a triple:

$$MT = (Tid, Wid, Aid),$$

where Tid is a transaction identifier, Wid is an identifier of the encompassing workflow, and Aid is an identifier of the activity to which MT is assigned.

Two multi-user transactions from two different workflows behave in the classical way, which means that they work in mutual isolation, and they are serialized by database management system. In case of access conflicts, resulting from attempts to operate on the same data item in incompatible mode, one of transactions is suspended or aborted, depending on the concurrency control policy.

Two multi-user transactions from the same workflow behave in a non-classical way, which means that the isolation property is partially relaxed for them. In case of access conflicts, the so called negotiation mechanism is triggered by DBMS, which informs users assigned to both transactions about the conflict, giving them details concerning the operations which have caused it. Then, using conferencing mechanisms provided by the workflow system, users can consult their intended operations and negotiate on how to resolve their mutual problem. If commonly agreed, they can derive a new version of a process instance, as described in Section 3.3, or merge their transactions, as proposed further in this section. In both cases, the users avoid future access conflict.

A particular mechanism is used in case of operations of the same multi-user transaction, if they are performed by different users, and they are conflicting in a classical meaning. There is no isolation between operations of different users, however, in this situation the so called notification mechanism is triggered by DBMS, which aims to keep the users assigned to the same transaction aware of the operations done by other users. We have to stress that it concerns only the situation when a user accesses the data previously accessed by other users, and the modes of those two accesses are incompatible in a classical meaning. After notification, users assigned to the same transaction continue their work, as if nothing happened. Notice, that

in case of the users assigned to the same activity, we assume not only strict collaboration, but also deep mutual confidence.

Now we focus on the operations which can be performed on multi-user transactions.

Every multi-user transaction is started implicitly by *initialize(Ti)* operation, which is performed by the system automatically at the very beginning of a corresponding activity execution, after the first database operation is requested by one of team members. This team member is called *transaction leader*. *initialize(Ti)* is also triggered automatically, directly after one of the team members has performed explicit *commit(Ti)* operation, or implicit *auto-commit(Ti)* database operation. All consecutive transactions of the same team are executed in a serial order.

After a multi-user transaction is initialized by the transaction leader, other team members can enter it at any moment of the transaction execution, by the use of explicit *connect(Ti)* operation, which is performed in an asynchronous manner. Once connected to the transaction, any member of the team can perform *disconnect(Ti)* operation, providing there is still at least one user assigned to this transaction. *disconnect(Ti)* operation breaks the link between transaction *Ti* and the user, who can next:

- close his session,
- suspend his operations for a particular time interval and re-connect to the same transaction later,
- wait until the transaction commits and connect to the next multi-user transaction of the same team,
- continue to work with different team in the scope of another multi-user transaction, provided he belongs to more than one team.

In cases: 1, 2 and 4, *disconnect(Ti)* operation plays the role of sub-commit operation, which means that the respective user intends to commit his own operations, and leaves the final decision whether to commit or not the multi-user transaction to his colleagues, whom he trusts.

Operations introduced up till now concern a single multi-user transaction. Next two operations: *merge(Ti)* and *split()* are special, since they concern two transactions. Transaction *Tj* can merge into transaction *Ti* by the use of *merge(Ti)* operation, providing the members of a team assigned to *Ti* allow for it. After this operation, transaction *Tj* is logically removed from the system, i.e. operation *abort(Tj)* is automatically triggered by the DBMS, and all *Tj* operations are logically re-done by transaction *Ti*. These actions are only logical, since in fact operations of *Tj* are just added to the list of *Ti* operations, and *Ti* continues its execution, however, the number of users assigned to it is now increased. It means, that until the end of *Ti* execution, the team assigned previously to *Tj* is merged into the team assigned to *Ti*. Of course, *merge(Ti)* operation is only allowed in the scope of the same workflow. *merge(Ti)* can be useful when an access conflict between two teams assigned to the same workflow arises.

Similarly to *merge* operation, *split()* operation can be used in order to avoid access conflicts (cf. Section 3). *split()* operation causes that a single multi-user transaction *Ti* is split into two transactions: *Ti* and *Tj*. After *split()* operation, a subset of team members, originally assigned to *Ti*, is re-assigned to newly created transaction *Tj*. Also all operations performed by re-assigned users are logically removed from transaction Ti and redone in transaction *Tj* directly after its creation.

Contrarily to merge operation which is always feasible, provided members of the other team allow it, split operation can be done only in particular contexts. Speaking very briefly, a transaction can be split if two sub-teams, which intend to separate their further actions, have operated on disjoint subsets of data, before split operation is requested. If the intersection between the data accessed is not empty, split operation is still possible, provided the data have been accessed by the two sub-teams in a compatible mode (in a classical meaning).

There is one more constraint concerning split operation. The resulting two transactions are related to each other in such a way that they can be either both committed or aborted. It is not possible to abort one of them and commit the other, since in this case the atomicity property of the original transaction (i.e. the transaction before split operation was requested) would be violated.

Finally, there are two typical operations on

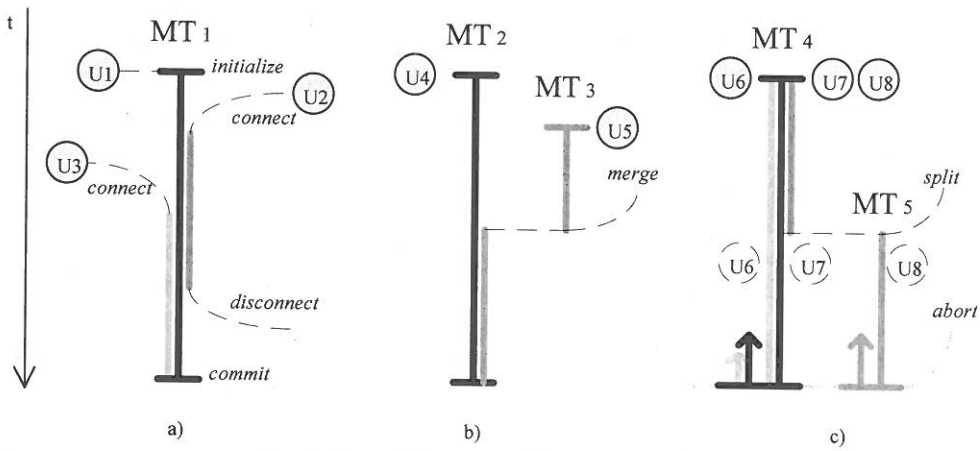

Fig. 4. Operations on multi-user transactions

transactions: *commit* and *abort*, which are performed in the classical manner.

All the operations presented above are illustrated in Fig. 4. Fig. 4a) shows *initialize, connect, disconnect* and *commit* operations, Fig. 4b) shows *merge* operation, while Fig. 4c) shows *split* and *abort* operations.

In our approach we use a multiversion database (cf. Section 2), which, taken as a whole, is generally inconsistent. Thus, we further refine the transaction definition in the following way: a multi-user transaction is a process that is addressed to a single database configuration and transforms it from one consistent state into another consistent state. We distinguish two multi-user transaction types: model and activity transactions.

Model transactions are used at the initialization and definition stages of a process life-time (cf. Section 2.2); they derive new versions of process models and/or update existing versions of process models.

Activity transactions are used during the process execution stage to perform operations described in a respective process model. An activity transaction corresponds exactly to a single activity of a process. It implies two important consequences. First, we assume that an activity, similarly to a transaction, is atomic, i.e. may not be performed partially. This restriction may be relaxed if we allow an additional level of process decomposition, namely tasks, as explained in Section 2.1. Second, the execution of a process which is composed of n activities corresponds to the execution of n transactions in the same database configuration. Some of them are serialized in the same way corresponding activities are serialized, while others may be executed concurrently.

Now, one can briefly discuss conflicts that may occur between transactions of the two types mentioned above. Two model transactions conflict very rarely because they are usually addressed to different process model versions, i.e. they are addressed to different database configurations. If, however, they are addressed to the same configuration and they access overlapping subsets of objects included in the process model, then a conflict may occur. There are no conflicts between a model transaction and an activity transaction, even if they address the same process model version. In this case (cf. Section 2.3), the model transaction is executed in the database configuration that is a parent of the configuration in which the activity transaction is executed. Also, two activity transactions concerning different versions of a process model or different instances of the same process model never conflict. The only possible conflict between transactions of this type is when they perform two activities of the same process instance that may be executed in parallel (cf. instance *i2* in Fig. 3).

To summarize, conflicts between transactions may occur if they are physically addressed to the same database configuration. They may

be easily resolved by transaction merge, as explained before. In the next subsection we show another mechanism of conflict resolution.

3.3. Conflict Versioning

Assume two activity transactions which are executed in the same database configuration and potentially conflicting (i.e. locking is necessary). What will happen if, after many operations on disjointed subsets of objects, those two transactions try to access the same object in an incompatible mode? In general, they are long-duration transactions, thus, aborting one of them is not recommended. The problem may be solved by providing a new database mechanism which is called conflict versioning.

Conflict versioning consists of the automatic derivation of a new private database configuration dedicated to a conflicting transaction (after detecting its first access conflict). The new private database configuration is a logical copy of its parent configuration which models process instance, with the exception that it contains all non-committed updates performed by the conflicting transaction, which are also logically removed from the parent configuration. Now, locking conflicts in the private configuration are no longer possible. If, because of some reasons (e.g. user request), the transaction aborts, the private configuration is simply deleted. Otherwise, i.e. if the transaction is commited, the system informs the user about the configuration derivation which becomes visible to other transactions.

After the transaction commitment, the activity schedule graph must be analyzed. If there is an activity (node) which requires artifacts from both separated subgraphs, then before starting it, the two database configurations must be merged. Otherwise, there is no need for merging and the process execution may be continued in two splitted configurations.

The conflict derivation technique is illustrated in Fig. 5. Transactions T_2 and T_3 are executed concurrently in the same database configuration, directly after the commitment of transaction T_1 (cf. Fig. 5a). When access conflict arises, a new database configuration is automatically derived (cf. Fig. 5b): execution of T_2 is continued in the parent configuration, while execution of T_3 is continued in the child configuration. When T_3 is commited, T_4 is initialized in the same configuration. Transaction T_5 requires artifacts from both subgraphs, which means that, after the commitment of T_2 and T_4, one must merge the two database configurations into a single one.

Two database configurations may be merged step by step by an object version comparison. This process, however, may be very tedious and usually requires many interactions with the user. Instead of this, a ***redo*** transaction may be performed. It consists of the automatic re-execution of operations performed in the child configuration once again in the parent database configuration. To support this mechanism, information about every transaction committed in the child configuration must be kept in a special file, usually called the log-file. During a ***redo*** operation, the contents of the log-file is interpreted and executed transaction by transaction in the parent configuration. After completing the last operation of the last transaction from the log-file, the ***redo*** process is finished and database configurations may be considered as merged. The ***redo*** operation is implemented as a single, multi-level transaction. Subtransactions nested in it correspond to transactions committed in the child configuration.

4. Experimental Evaluation

The concepts presented in Section 3, in particular: multi-user transactions, negotiation and notification mechanisms, conflict versioning, have already been experimentally evaluated in *Agora* prototype. *Agora*[1] is an asynchronous collaborative Web-based software prototype which is built on the top of commercial relational DBMS. *Agora* is composed of two main parts. The first one is a conferencing tool allowing negotiations between cooperating partners. The second is a collaborative document writing tool allowing edition of electronic documents. The database, being a kernel of the system, is used to store both negotiation history and collaboratively written documents. *Agora* is independent of hardware, operating systems, browsers and database management systems, accessible to any Internet user under condition of proper registration.

[1] Agora is available at the following URL: http://aquila.kti.ae.poznan.pl:1664/Agora

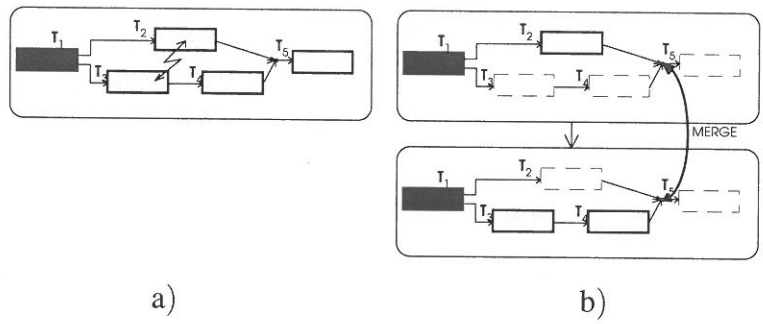

Fig. 5. Conflict Derivation

A single conference in *Agora* corresponds to a single multi-user transaction, thus participants of the same conference constitute a single team working on the same document. *Agora* provides conference participants (negotiators) with an arbitrary number of conferences and arbitrary number of collaboratively written versionable documents, with the restriction that only one document can be associated with a single conference. All conference participants discuss and present their positions by exchanging electronic messages. Each participant of a conference sees all the messages exchanged. Every user can be involved in several negotiations simultaneously, i.e. he can virtually attend different conferences. Negotiations in different conferences may concern different topics, different aspects of the same topic, or the same topic discussed by different partners. Notice, that in terms used in Section 2, a single conference corresponds to single activity (currently document writing) which is performed collaboratively by a team of users, being conference participants.

The part of *Agora* devoted to support collaborative writing is required to prepare the final document which is the result of negotiations. This common document is seen by and accessible to all conference participants. When a conference participant writes or modifies a paragraph of the document and confirms the changes, it becomes instantaneously visible to other participants. Next, any participant can modify this paragraph. *Agora* window used for the purpose of collaborative document writing is illustrated in Fig. 6.

Agora has been implemented in Java language and connected to the *Oracle* database management system through Java Database Connectivity interface (JDBC) to provide persistency of both documents and negotiation history. The use of Java and JDBC provides Agora with platform independence, concerning hardware, operating systems and database management system.

The structure of *Agora* client and server is presented in Fig. 7. The main part of a client is *Agora Client Applet*, which operates on Java Virtual Machine (JVM). It is accessed through Internet by the use of a standard WWW browser. The main part of the server is *Agora Server Kernel* that also operates on Java Virtual Machine. It is directly accessible through Internet. It uses Java-Database Connectivity (JDBC) interface to access Oracle DBMS.

Concurrency control mechanisms provided by Oracle DBMS are overridden in *Agora Server*, what is necessary to validate the concept of multi-user transactions and new concurrency control mechanisms proposed in this paper.

Every document version is stored in one database table, thus if a document is available in n versions, then n tables have to be created. The first version of a document is entirely represented in a respective table, while in case of derived document versions only differences in comparison to the parent document version are represented, i.e. paragraphs explicitly modified in the child document version. This aims to avoid redundancy which can be really painful in case of documents having many slightly different versions.

Every paragraph of a document version is stored in a single raw of a corresponding table, which is composed of a paragraph content and its layout attributes. Paragraph content is modeled by a single attribute of long raw type. It means that a paragraph can contain not only pure text but also multimedia data (pictures, sounds, etc.). Layout attributes contain typical information about

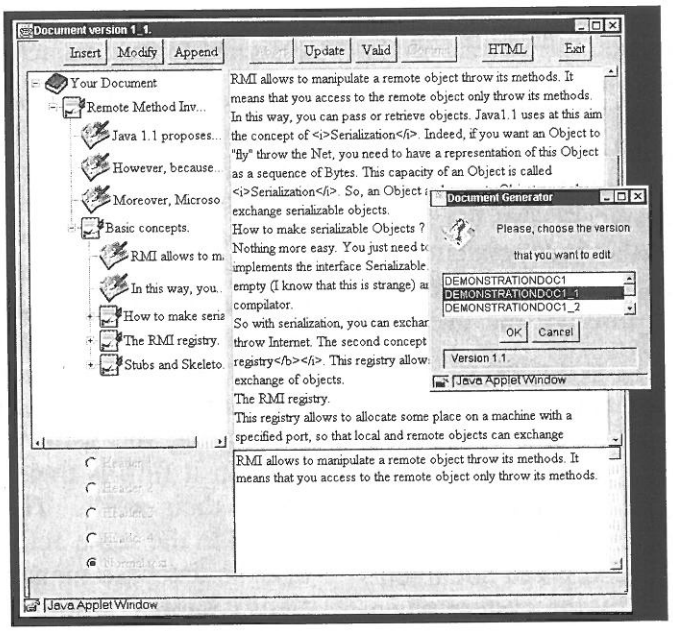

Fig. 6. Collaborative document writing

the way a paragraph is visualized to the users, e.g. color, font, size, indent.

To summarize, let us briefly compare *Agora* with other CSCW systems, putting a particular emphasis on what distinguishes *Agora* from them. Almost every CSCW system (Crowley et al. 1990, Ellis et al. 1991, Ensor et al., Garfinkel et al., Hill et al., Stefik et al.), including *Agora*, provides flexible message exchange between users who are grouped in a way reflecting the cooperation structure. Typically, the notion of a conference or a discussion group is used, which additionally supports the cooperating users, offering them a variety of asynchronous and synchronous tele-conferencing tools and mechanisms. Similarly to the *Agora* system, these tools and mechanisms, provide the users with mutual awareness, notification and negotiation support. Taken together, they make an illusion for the users of working in the same virtual room. Moreover, most of CSCW systems offer shared white boards and shared documents. Shared white boards facilitate the discussion between users, giving them a possibility of visualization or illustration of some concepts and ideas. Shared documents support the collaborative edition of written materials which are final outputs of conferences.

Most of the available CSCW systems are platform-dependent. They are bound to particular software environments and network architectures. On the contrary, *Agora* is a quite open system and platform-independent system, since it is written in Java language. Thus, it may be accessed by anyone who is connected to the Internet and runs web browser or Java applet viewer.

Many CSCW systems (e.g. Lotus Notes) pro-

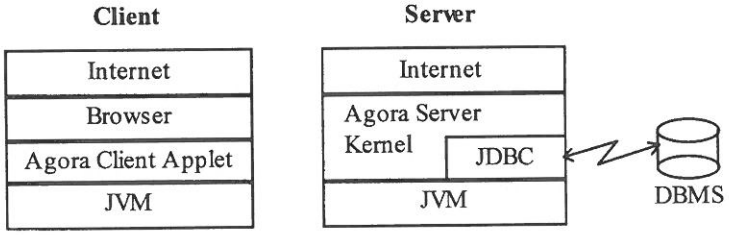

Fig. 7. Agora client and server structure

vide typical database functions, like object persistency, access authorization, concurrency control. These functions are, however, implemented from scratch. *Agora*, contrarily to those systems, uses commercial RDBMS. It makes *Agora* very efficient and reliable, since the database technology is very mature and has been verified over almost thirty years. It is worth emphasizing that *Agora* is also independent from the RDBMS being used, because it communicates with the database through the universal JDBC interface.

Finally, there are some features of the *Agora* system which definitely distinguish it from other CSCW systems. First, in the *Agora* an original approach to document versioning has been implemented (Wieczerzycki 1996, Wieczerzycki 1998), in which new versions of documents are automatically derived by the system whenever there is a problem of data access. Second, since the transaction model of a commercial RDBMS is too restrictive for the requirements of cooperation, the *Agora* overrides the transaction model of RDBMS and implements a new one. This new transaction model, as presented in the paper, is addressed to groups of strictly collaborating users.

5. Conclusions

The main goal of this paper was to propose a flexible, persistent environment for workflow system applications which enables efficient business process modeling and execution. On one hand, a special emphasis has been put on the dynamic evolution of process description over a given time, and on the support for cooperative activities performed by teams of users, on the other.

The basic idea of the proposed approach is to extensively use object versions which are always considered in particular contexts, called configurations. The configuration is both a granule of database versioning and a unit of database consistency. When a business process is modeled, configurations substantially simplify the representation of its historic versions, which may be useful in the near future, and current alternative versions (variants), which must be kept in the database simultaneously. When a business process is executed, configurations support the avoidance and the resolution of conflicts arising between concurrently executed transactions assigned to process activities.

Another important idea of the proposed approach is to use multi-user transactions and associate them with activities embedded in the processes. We feel that the proposed transaction model has many advantages. First of all, it is very straightforward and not complex, thus the management of transactions of this type does not cause substantial problems. Second, the proposed transaction model allows practically unrestricted collaboration among people performing the same activity. As a consequence, it fulfills their requirements and simplifies their work. Third, since the users assigned to the same activity preserve their identity, the database management system can efficiently support users' awareness and notification, which are two very important functions of every collaborative system. Finally, the proposed model is very close to the classical ACID transaction model. As we have mentioned before, it makes the model more reliable and easy to implement, since classical ACID transactions dominate at the commercial database market.

It should be emphasized that both the transaction model and transaction management mechanisms have been elaborated parallelly with the development of the prototype collaborative system, called *Agora*. Thus, the proposed approach is not purely theoretical, but instead, it reflects the problems and solutions which occurred during the implementation of *Agora*.

References

AIELLO L., NORDI D., PANTI M., *Modeling the Office Structure: A First Step Towards the Office Expert System*, Proc. of 2nd ACM SIGOA Conf., 1984.

BENFORD S., *Requirements of Activity Management*, in *Studies in CSCW: Theory, Practice and Design*, North–Holland, 1991.

BRIERLEY E., *Workflow Today and Tomorrow*, Proc. of Conf. on Document Management, 1993.

BUCHMANN A., OZSU M.T., HORNICK M., GEORGAKOPOULOS D., *A Transaction Model for Active Distributed Object Systems*, in: Elmagarmid A. (ed.), *Advanced Transaction Models*, Morgan Kaufmann, 1992

BULLINGER J.H., MAYER R., *Document Management in Office and Production*, Nachrichten fur Dokumentation, Vol. 44, 1993.

BUSSLER C., JABLONSKI S., *Implementing Agent Coordination for Workflow Management Systems Using Active Database Systems*, Proc. 4th Int. Workshop on Research Issues in Data Engineering: Active Database Systems, 1994.

CHRYSANTHIS P.K., RAMAMTITHAM K., *Acta: A framework for specifying and reasoning about transaction structure and behavior*, Proc. of ACM-SIGMOD Int. Conference on Management of Data, 1990.

CROWLEY T., MILAZZO P., BAKER E., FORSDICK H., AND TOMLINSON R., *MMConf: An Infrastructure for Building Shared Multimedia Applications*, Proc. of ACM Conference on Computer Supported Cooperative Work, October 1990, pp. 329–342.

ELLIS C.A., GIBBS S.J., AND REIN G.L., *Groupware: Some Issues and Experiences*, CACM 34:1 (January 1991), pp. 38–58.

ELMAGARMID A. (ed.), *Database Transaction Models*, Morgan Kaufmann, 1992.

ELAMAGARMID A., LEU Y., LITWIN W., RUSINKIEWICZ M., *A Multidatabase Transaction Model for Interbase*, Proc. of VLDB Conf., Brisbane, 1990.

ELIASSEN F., VEIJALAINEN J., TIRRI H., *Aspects of transaction modeling for interoperable information systems*, in: Interim Report of the COST 11ter Project, 1988.

ENSOR J.R., AHUJA S.R., HORN D.N., AND LUCCO S.E., *The Rapport Multimedia Conferencing System: A Software Overview*, Proceedings of the 2nd IEEE Conference on Computer Workstations, March 1988, pp. 52–58.

GARCIA–MOLINA H, SALEM K., *Sagas*, Proc. of the ACM Conf. on Management of Data, 1987.

GARFINKEL D., WELTI B., AND YIP T., *HP Shared X: A Tool for Real-Time Collaboration*, Hewlett-Packard Journal, April 1994, pp. 23–24.

GEORGAKOPOULOS D., *Transactional Workflow Management in Distributed Object Computing Environments*, Proc. 10th int. Conf. on Data Engineering, 1994.

GRAY J., *Notes on Database Operating Systems*, Operating Systems: An Advanced Course, Springer–Verlag, 1978.

HALES K., Workflow Management. An overview and some applications, Information Management and Technology, Vol. 26, 1993.

HAWRYSZKIEWICZ I., T., *A Generalized Semantic Model for CSCW Systems*, Proc. of 5th Int. Conf. on Database and Expert Systems, Greece, 1994.

HENDLEY T., *Workflow Management Software*, Information Management and Technology, Vol. 25, 1992.

HENNESSY P., BENFORD S., BOWERS J., Modeling Group Communication Structures: An Analysis of Four European Projects, In SICON 89: Proc. of the Singapore Conf. on Networks, IEEE Press, 1989.

HILL R., BRINCK T., ROHALL S., PATTERSON J., AND WILNER W., *The Rendezvous Architecture and Language for Constructing Multiuser Applications*, ACM Transactions on Computer Human Interaction 1:2 (June 1994).

HUHNS M., N., SINGH M.,P., *Automating Workflows for Service Provisioning: Integrating AI and Database Technologies*, Proc. of 10th Conf. on Artificial Intelligence for Applications, 1994.

JONES J.I., MORRISON K.R., *Work Flow and Electronic Document Management*, Computers and Industrial Engineering, Vol. 25, 1993.

KLING R., *Cooperation, Coordination, and Control in Computer Supported Cooperative Work.*, Comm. ACM, Vol 34, No 12, Dec. 1991.

LAVERY M., *A Survey of Workflow Management Software*, Proc. Conf. on OIS Document Management, 1992.

MCCLATCHEY R., BAKER N., HARRIS W., LE GOFF J–M., KOVACS Z., ESTRELLA F., BAZAN A., LE FLOUR T., *Version Management in a Distributed Workflow Application*, Proc. of 8th Int. Workshop on Database nad Expert System Applications — DEXA'97, France, 1997.

MEDINA–MORA R., WINOGRAD T., FLORES R., FLORES F., *The Action Workflow Approach to Workflow Management Technology*, Information Society, Vol. 9, 1993.

MOSS J. E., NESTED TRANSACTIONS: AN APPROACH TO RELIABLE DISTRIBUTED COMPUTING, The MIT Press, 1985.

NODINE M., ZDONIK S., *Cooperative transaction hierarchies: A transaction model to support design applications*, Proc. fo VLDB Conf., 1984.

REUTER A., Contract: A means for extending control beyond transaction boundaries, Proc. of 2nd Workshop on High Performance Transaction Systems, 1989.

RUSINKIEWICZ M., SHETH A., *Polytransactions for managing interdependent data*, IEEE Data Engineering Bulletin, 14(1), 1991.

SCHUSTER H., JABLONSKI S., KIRSCHE T., BUSSLER C., *A Client/Server Architecture for Distributed Workflow Management Systems*, Proc. of 3–rd Int. Conf. on Parallel and Distributed Information Systems, 1994.

SHETH A., *Transactional Workflows: Research, Enabling Technologies and Applications*, Proc. 10th Int. Conf. on Data Engineering, 1994.

SMITH H., HENNESSY P., LUNT G., *The Activity Model Environment: An Object-Oriented Framework for Describing Organizational Communication*, Proc. ECSCW Conf., 1989.

STEFIK M., FOSTER G., BOBROW D.G., KAHN K., LANNING S., AND SUCHMAN L., *Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings*, CACM 30:1 (January 1987), pp. 32–47.

VOSSEN G., WESKE M., WITTKOWSKI G., *Dynamic Workflow Management on the Web*, Fachbericht Angewandte Mathematic und Informatic 24/96–I, Universitat Muenster, 1996.

WIECZERZYCKI W., *Advanced Transaction Management Mechanisms for Document Databases*, Int. Workshop on Issues and Applications of Database Technology — IADT'98, Berlin, Germany.

WIECZERZYCKI W., *Versioning Technique for Collaborative Writing Tools*, Proc. of 7th International Conference and Workshop on Database and Expert Systems Applications DEXA'96, IEEE Computer Society Press, Zurich, Switzerland, September 1996, pp. 463–468.

Received: July, 1996
Accepted: March, 1998

Contact address:
Waldemar Wieczerzycki
Department of Information Technology
University of Economics at Poznań
Mansfelda 4, 60–854 Poznań
Poland
Phone: (48) 61 848.05.49
Fax: (48) 61 848.38.40
E-mail: wiecz@kti.ae.poznan.pl

WALDEMAR WIECZERZYCKI received the M.Sc. degree in Computer Science from the Technical University of Poznan in 1983, and the Ph.D. degree in Computer Science from the Technical University of Gdansk in 1992. From 1984 to 1992 he was with the Institute of Computing Science at the Technical University of Poznan. From 1992 to 1996 he was an Associate Professor at the Franco-Polish School of New Information and Communication Technologies in Poznan. Since 1996 he has been with the Department of Information Technology at the University of Economics at Poznan.

He participated in several industrial projects concerning real-time operating systems, database applications, business process reengineering and Intranet applications. His research interests include object-oriented and deductive databases, in particular concurrency control, complex management objects, and version support and management. Recently he has been also interested in CSCW applications, in particular workflow modeling, collaborative writing and collaborative software design.

He is the author of 4 books and 45 papers in international journals and conference proceedings.