In Proc. of the 5th Australasian Conference on Information Systems, Melbourne 1994

A Formal Approach to Synchronize Long-lived Computations

Friedemann Schwenkreis

Institute of Parallel and Distributed High-Performance Systems (IPVR)
University of Stuttgart, Breitwiesenstr. 20-22, D-70565 Stuttgart, Germany
eMail: schwenk@informatik.uni-stuttgart.de

Abstract
Since long-lived computations (e.g. workflow systems) cannot be realized with classical transactions, extended models have been developed to support this special class of executions with proper semantics. The main problem with these new models is the lack of an extended concurrency control mechanism which makes it possible to use all advanced features of the new methods. This article presents a formal approach for a sychronization method which is powerful enough to be used in all extended transaction models and which can be used for new working paradigms as e.g. CSCW.

INTRODUCTION

Workflow and computer-supported cooperative work (CSCW) are nowadays very important keywords. Nearly every software vendor has to offer solutions for these areas, but there is a severe problem with all these "solutions": The transactional concept cannot be used to model these executions and therefore we loose the well defined behavior of this very helpfull abstraction.

Although new "extended" transaction models have been developed (Elmagarmid 1992) to support the advanced applications with proper semantics, serializability is still the correctness criterion in the field of concurrency control which is insufficient for these advanced transaction models. Furthermore, most of the concurrency control mechanisms make the assumption that there is a global instance (scheduler) which has the control and the knowledge about all accessing operations in a system. Especially in modern distributed systems, this assumption is very hard to realize and in federated systems with local accesses this approach makes no sense at all.

The model for concurrency control presented in this article is object-based and not a history oriented. We will introduce a new correctness criterion and the necessary object criterions. Additionally, the necessary protocol for a concurrency control mechanism is described and a short discussion of the advantages of the approach is included. The comparison with serializability-based methods beyond the scope of this paper. This topic will be discussed in a different article (Performance of Concurrency Control Mechanisms in Centralized Database Management Systems, Prentice Hall).

AN EXECUTION MODEL

Before introducing an approach for controlling concurrency in case of long running computations we first have to define an execution model for these computations. This section will introduce such a basic model which is similar to the ConTract model (Waechter et al. 1992) but has some extensions.

Basic elements

This subsection presents several useful definitions on which the model of long computations' control flow is based.

Definition 1: The *context* is a set of data objects. These data objects are only accessible from the computation in which they are defined. Furthermore, they are stored in an update free way in stable storage (i.e. every update creates a new object version).

Definition 2: An *activity* $a_i \in A$ is a basic unit of work for which the ACID properties (atomicity, consistency, isolation, durability), similar to classical transactions (Bernstein et al. 1987), are guaranteed. The result of these activities is either "COMMIT" or "ABORT". There is a special activity set $M \subset A$ which consists of so-called *management activities.* These management activities only access objects from the context and are able to return a boolean value in addition to the normal results mentioned above.

Definition 3: A *final state* $f \in F$ is a state of the computation which signals the end of the execution.

Definition 4: There is a set of *events* E in which we differentiate between two categories. *External events* $e_i^e \in E$ are triggered from outside the computation like *event*(*external*, *Start_-Event*). *Internal events* $e_i^a \in E$ are the results of the execution of an activity a_i and can be described as: *event*(*activity*,*result*) (e.g. *event*(a_i , *COMMIT*)).

Definition 5: A *transition* $t_i \in T$ is a pair (p, w) $w \in A \cup F$ where p is a predicate of the form $(p_1 \vee p_2 \vee ... \vee p_i)$ $i \geq 1$ and p_i is a set $\{e_i \,|\, e_i \in E\}$ which represents a conjunction of the form $(e_k \wedge e_l \wedge ...)$. This means that an activity is started (or the end of the computation is reached) if one of the event combinations $(e_k \wedge e_l \wedge ...)$ happens.

Definition 6: There is a *path* from an activity $a_i \in A$ to an activity $a_j \in A$ written $<a_i ,a_j>$ if there exists a transition $t = (p,a_j)$ where $\exists\, p_k \in p \;\; e_l \in p_k \;\wedge\; e_l = (a_i, result)$. If there is a path $<a_i ,a_j>$ and a path $<a_j ,a_k>$ then we state that there is a path $<a_i ,a_k>^+$ (transitivity). If there exists a path $<a_i ,a_j>^+$ and a transition (p, f) $f \in F$ and $\exists\, e_i \in p_k \;\; e_i = event(a_j, result)$ then we state that there exists a path $<a_i , f>^+$ (similar for external events in case of the start of a path).

Constraints and behaviour at run time

With the basic elements presented in the previous subsection we are able to describe the control flow for a computation. A process definition n (see below) is wellformed if:

- $\exists\, t \in T \;\; \exists\, p_i \in p \;\;\; p_i \equiv \{event(external, Start\_Event)\}$:
  A transition which reacts on the start of the computation is needed.
- $\exists\, t \in T \;\;\; t = (p,w) \wedge w \equiv End\ (End \in F)$:
  A transition which ends the computation is needed.
- $\forall a_i \in A \;\; \exists <a_i, End>^+$:
  A path to the end of the computation exists for every activity.
- $\forall a_i \in A \;\; \exists <event(external,Start\_Event), a_i>^+$:
  Tere is a path form the start event to every acitity.

- The definition of the computation must avoid any internal events after a final state is reached.

These requirements are necessary but not sufficent. The correctness of a computation defined with the mechanisms above cannot be proven at definition time.

Definition 7: A *process definition n* is a tupel of sets $(\{o\},\{a\},\{t\},\{e\},\{f\})$ where: $\{o\}$ is a context, $\{a\} \subset A$, $\{t\} \subset T$, $\{e\} \subset E$, $\{f\} \subset F$.

In the following we take a closer look at the way a process definition n can be interpreted at run time which leads to the criterion for correctness. We assume that the process definition introduced above is instantiated (written $\tilde{n}$) at run time and used as a template for the computation:

- If the start event for a computation is triggered, an instance $\tilde{a}$ for every activity is generated which is triggered by the corresponding transition. An instance $\tilde{t}$ is generated for every transition in the definition which references the corresponding activity in its predicateand every a is replaced by its instance $\tilde{a}$.
- If an internal event is triggered by an activity instance $\tilde{a}$ (e.g. event($\tilde{a}$, COMMIT)), all transition instances $\tilde{t}$ which reference the instance are checked and if the event is element of any p_j of this transition it is replaced by "TRUE" and p_j is evaluated.
  If all events $e \in p_j$ are replaced by "TRUE" the coresponding activity w, is instantiated and so are all transitions which reference this activity (see above).
  If the transition instances already exist, only the activity is replaced by its instance.
- Activities which have triggered an event and transition instances which have triggered are removed from the system.
- If a final state f is triggered by a transition, the instance of the process definition is removed from the system.

If we have an executing system which follows the protocol above, we can define a history of a computation.

Definition 8: A *computational history* $h_{\tilde{n}}$ is a sequence of activity instances $\tilde{a}$, transition instances $\tilde{t}$, events e, and final states f which appeared during the execution of a process instance $\tilde{n}$.

It is easy to extend this definition to the history of the whole system H and therefore we do not present the definition here.

Recovery issues

Similar to the approach of ConTracts the run time system guarantees the reliable execution of a process definition. This means that in case of a failure there is no automated rollback of the whole computation. Instead, only active activities (which have not triggered an event) are aborted and the execution is continued (forward recovery) by restarting the aborted activities.

What happens if the user aborts the execution? In this case, the user wants the execution to be undone. Since we do not have the protection of a transaction for the whole exectuion, we cannot use a log-based undo mechanims. Instead, we have to introduce a compensation mechanism (Gray 1981, Garcia-Molina 1983, Waechter et al. 1992):

Definition 9: For each activity a there must be a so-called *compensation activity* $c \in C$ which is called the (semantical) counteraction of a.

There are several approaches to define the execution sequence of compensation activities if the compensation has to be run. One approach is to define the control flow explicitly with the compensation included. Another approach is that the executing system controls the compensation on a timestamp bases. We will not go into further details at this point, because our main issue is the concurrency control and not the control flow. Nevertheless we need to define what is happening if the "compensating path" is entered.

Definition 10: There is a special external event $e^e \in E$ called *Compensate_Event*. The system reacts on this event by removing all transition instances $\tilde{t}$, aborting all activity instances and by executing compensating activities c_i for all activity instances $\tilde{a}_i$ which triggered an event ($\tilde{a}_i$, *COMMIT*). If all necessary c_i are executed a final state f called "Unsuccessful_End" is reached.

The definition above makes save that after the "Compensate_Event" no further activity of the "normal" process definition is executed. This means that we do not take partial compensation (after which the execution proceeds) into account.

Correctness

In the previous subsections we have introduced a mechanism to describe and execute computations, now we have to investigate their correctness. Our view of correctnes is very close to Garcia-Molina's (1983), because we make nearly the same assumptions.

- A state without active computations is consistent.
- Every activity a transforms a consistent state into a consistent state (transactional).
- If a computation starts in a consistent state it ends in a consistent state when it reaches a final state f (if the process definition is correct).
- While a computation is running be inconsistent (intermediate) states (from the application's view) might occur.

From the statements mentioned above we can derive the criterion for a correct synchronization mechanism. Since the computation is based on transactional activities, we do not need any new correctness criteria or concurrency control on the activities' level (see also stepwise serializability in Garcia-Molina 1983).

Definition 11: :A history h is *correct* if:

a) The order of the activity instances $\tilde{a}$ appearing in $h_{\tilde{n}}$ satisfies $\tilde{n}$

b) $\forall_{\tilde{n}}: f_{\tilde{n}} \in h$ or
$\forall_e \; e = (\tilde{a}_i, COMMIT) \wedge e \in h$: c_i can be executed successfully

In other words the system history is correct if the process histories contain a final state (i.e. the execution has reached its end) or if it is guaranteed that all acitve executions can execute at least their compensational activities. This is seen only from a data object view which does not include system failures. More precisely, it must be ensured that for the (global) data objects a state is maintained on which the compensating activities could be executed.

THE CONCURRENCY CONTROL PROBLEM

The task of concurrency control is to ensure the correct execution of computations running in parallel ("first law" of concurrency control Gray et al. (1993)). If we take a closer look at definition 11, we see that point a) does not needs concurrency control. Only the second criterion of point b) must be taken into account to ensure correctness. This section discusses the possibilities to realize a proper mechanism and how the same mechanism can be used to increase the probability that a successful end of a computation can be reached.

Ensuring compensation

As mentioned before, the only conflict between computations running in parallel occurs when one computation changes the object state in a way that compensating activities of the other computation cannot be executed any more. For example, a computation p transfers money from an account a at bank A to an account b a bank B. The process definition for this example includes two activities. One activity w to withdraw some money from the account at A and a second activity d to deposit the money on the account b. The corresponding compensation activity for w is c which deposits the money on account a.

After the commitment of w we have to to make sure that c can be executed (definition 11). If we do not take any measures to ensure this the following could happen:

- w withdraws money from a and commits
- An activity from a concurrent computation q closes account a
- The user which started w recognizes a mistake and triggers a "Compensate_Event" for n.
- The system tries to execute c but fails, because a has been closed.

After this scenario there is no way for p to reach a final state, i.e. to reach a consistent state. This is one example for problems which can occur if there is no mechanism to protect data objetcs after they have been accessed by a computation.

The classical solution to solve the concurrency control problem is to use locks on the objects involved in a computation. The main problem with this approach (which stems from the serializability criterion (Bernstein et al. 1987)) is that it restricts the access to the objetcs and therefore it unnecessarily restricts the possible parallelism. Especially in long-running executions, this results in a high probability of deadlocks and long blocking times or high transaction abort rates. Furthermore, a system which supports locks outside transactions does not exist and the computations we presented are not transactions at all.

The main drawback of the classical locking approach is the automated handling of locks and therefore the low differentiation between lock modes (shared / exclusive). In our example, we see that an existence lock of the account a would be sufficient to ensure the compensation. This would allow concurrent computations to change the account in every way they want.

Looking at an example in which the original activity deposits money and the compensation activity withdraws the money, we find a more complex characteristic. In this case we have to ensure that the amount of money must be available on the account until the computation has reached a final state.

A more severe problem in our case is that the set of objects required by compensation activities must not be the same or a subset of the set of objects accessed by the coressponding activity. A good example for this case is a process description for a production process. The first activity *a* produces some parts which are needed for an assembling activity *b*. The compensation activity *c* for activity *a* cannot undo the production. Instead, the produced parts have to be put in the stock and the corresponding database entries must be updated. In the normal (noncompensating) process the object "stock" is never accessed.

All the above examples show the need for a mechanism to exploit the semantics of activities. The following will go into details of methods how to define and manage the semantical information of activities.

Support for successful executions

Apart from the correctness of executions, a second criterion for concurrency control mechanisms exists: the throughput (second law of concurrency control (Gray et al. 1993)). All features discussed do not take the successful execution into account. From the point of correctness it is sufficient to be able to reach any consistent state (including the unsuccessful execution). Om the contrary, the user of a system is interested in the amount of successful executions he can execute in a certain time. Therefore, we need a mechanism which supports the executability of activities similar to mechanisms for the compensating activities.

A simple solution for the problem to guarantee the successful execution of a process definiton would be to preclaim all required resources and to reserve them before starting the execution. It is easy to see that this approach is more restrictive than a locking method and has the same and even more drawbacks. Especially in the field of declarative query languages (e.g. SQL) and with the usage of addressing by run time variables this is an unusable method because it is unknown which objects are involved in an execution.

Again, we have the problem that the executing system does not know enough about the semantics of the activities. Furthermore, we have a problem very similar to the correctness ensurance because we need to protect resources (data objects) from concurrent access to be able to continue our execution. The only difference to the correctness problem is that if an activity is executed all objects which are needed for compensation should be known and can be protected. In case of the "normal" execution we do not know which resources are needed e.g. for the fifth activity if we have executed the first activity.

Because the problem of ensuring correctness can be seen as a subset of the problem to ensure the successful execution, we do not differentiate in the following between them and subsume them under the term concurrency control problem.

SPHERES OF ISOLATION

This section will introduce approaches for solving the concurrency control problem described above. Similar to the "Spheres of Control" (Davies 1978) a model for isolating computations is presented.

Extreme approaches

Before we go into the discussion of methods we need a proper abstraction of the concurrency control problem introduced before.

Definition 12: A *data object* $o \in O$ is a tupel (n_o, v_o, s_o) where n_o represents the name, v represents the value and s the represents the status of the object which means information about the accesses on the object.

The definition of data objects is different from classical once. Aside the name and the value (which are well known attributes of objects) we introduce a status of a data object. We assume that the status holds information about the last access performed on the object. For example in a transactional environment this could be the transaction identification or information about the user who accessed the object. The status must be updated by every accessing operation.

Definition 13: A *state* is the set of all objects (the representing tupels (n_o, v_o, s_o)) at the same time and with z we denote the current system state. States on which an activity $a_i \in A \cup C$ can be executed successfully are called *acceptable* states of a_i: $accept(a_i)$. The intersection of the acceptable states of all (normal) activities are called *wanted states* $Z_a = \cap\, accept(a_i)$, $a_i \in A$. All *possible correct states* are the intersection of the acceptable states of all compensating activities in the system: $Z_c = \cap\, accept(c_i)$, $c_i \in C$.

With the previous definitions it is easy to define when the computations in the system can finish their work successfully and the correctness is ensured:

Definition 14: A system state z is *successful finishable* iff $z \in Z_a \cap Z_c$.
And corollary: a system state is *correct* iff $z \in Z_c$.

A severe problem of a process definition is to ensure that $Z_a \cap Z_c \neq \emptyset$. Otherwise we reach a point in the execution where we cannot proceed. The goal of a concurrency control method in our model must be to ensure correctness and to ensure as far as possible that the system state changes only to successfully finishable states. As we have stated above, it is impossible to know the set Z_a before an execution starts and therefore we can decide only at the execution time of computations if a state change caused by an activity violates the successful finishability or the correctness. Furthermore, it is impossible to evaluate the whole system state if changes are made by an activity. Instead, we have to find a mechanism which has to evaluate only a minimal subset of the system state (the "necessary" objects). Additionally, we have to develop mechanisms which manage an object view of system states.

Definition 15: An *object view* Z^o from object o of a set of system states Z is the set of tupels (n,v,s) with the same object name n_o in all elements of Z:

$$Z^o = \cup\, (n_o, v_i, s_i) \text{ , } (n_o, v_i, s_i) \in X \wedge X \in Z.$$

With the definition of an object local view of a system state we are now able to define the correctness from the view of an object:

Definition 16: From the view of an object o its state is *correctness supporting* if $(n_o, v_o, s_o) \in Z_c^o$.
And similar: it is *success supporting* if $(n_o, v_o, s_o) \in Z_a^o$.

Now the question is whether the correctness supporting criterion for all objects guarantees that the system is always in a correct state. Unfortunately, the answer is no. A simple example can show the insufficiency of the criterion: Think about a compensation activity c which has access to two accounts A and B and the semantics is that the activity has to pay a bill of amount c. The criterion for a correct system state is that $A.v + B.v \geq c$. This results in several correct states dependent on the distribution of the money, e.g. $A.v = 0.5 * c \wedge B.v = 0.5 * c$ or $A.v = c \wedge B.v = 0$. A state where $A.v = 0.5 * c$ and $B.v = 0$ would be a correctness supporting state for both objects but an incorrect state of the system.

The discussed approaches are both extreme cases. The first was a criterion for all objects and the second a pure object local one. As a conclusion, we can state that a senseful solution of our concurrency problem can only be found in between these two extremes.

Connecting objects

To find a practicable solution for the concurrency problem we start from the object local view and try to expand this view over several objects until a correct state is ensured.

Definition 17: A *virtual object* $k \in K$ is a combination of two objects $o, u \in O \cup K$ represented by a tupel (n_k, v_k, s_k) where r_k is a computation rule $v_o \otimes v_u$ (v_o can be replaced by a rule) and $s_k = \emptyset$. $\otimes$ is called *combination operator*. Additionally there is an evaluating function $val_k(r_k)$ which computes the rule r_k with all actual values of the referenced objects.

The definition above allows us to group objects together under a virtual object which represents the connected objects. We can reuse the above example: two accounts A and B can be grouped to a virtual object called account R with the rule $r_R = v_a + v_b$. If a third account C can be used, the virtual object R can be grouped with this account to a virtual object called account S with the rule $r_S = v_a + v_b + v_c$

The next step is to take the virtual objects into account in our state definition:

Definition 18: An *extended state* is the set of all tupels (n_o, v_o, s_o) representing all objects o (at a specific point of time), united with the tupels $(n_k, val_k(r_k), s_k)$ of all virtual objects k.

All definitions for sets of states will be used with this new state definition. Due to limited space we do not explicitly present the new definitions. There is only one really important criterion which results directly from our new view of states:

Definition 19: From the view of an object o, its state is *extended correctness supporting* if:
$(n_o, v_o, s_o) \in Z_c^o \;\wedge\; \forall_{k \in K} \textit{ where } \exists\, r_k \textit{ with } v_o \in r_k\!: (n_k, val_k(r_k), s_k) \in Z_c^k$
(Similar for the definition of *extended success supporting*).

In other words, for extended correctness support it is necessary that not only the object itself is in a state where all compensation activities could be run. Additionally, every virtual object in which the object is referenced has to be in a correctness supporting state.

From a global view, we are now able to consider not only object local criterions. Instead, we can group objects under a virtual object to use the knowledge of several objects. This will lead us to concurrency control mechanisms which allow much more parallelism than methods only using object local information.

A concurrency control mechanism

Based on the definition for extended correctness support of objects (definition 19) we are able to develop a mechanism for controlling concurrency on an object basis. Therefore, we first need to introduce a method to manage an object's set of possible correct states Z_c^o (the same for the set of acceptable states Z_a^o) which is the real new feature in our approach.

It is clear that we cannot manage this set directly, because in general there is an infinite number of tupels representing the allowed states of an object. Instead, we have to use a declarative representation. There are two ways to introduce this declarative notion:

- A so-called *positive* approach in which expressions are used to declare which states are valid. This means that we must explicitly declare all allowed states and initially all states are invalid.
- A so-called *negative* approach in which expressions are used to declare which states are forbidden. Thsi means that we must use an expression to disallow states and initially every state is allowed.

We use the negative approach in our mechanism because it is also used in relational database systems and integrated in the SQL2 standard (constraints: Melton et al. (1993)).

We assume that all constraints on an object are stored locally at the object and can be seen as an extension of the object.

Definition 20: For every object there exists a *set of constraints* $B^o = \{b_i\}$ which represents a condition which must hold for the object. All constraints are classified by the identification of the constraint's establisher and a qualifier which indicates if the constraint is for correctness or success support.

From the object's view an access can change the object in the following ways:

- The name is changed (e.g. by a delete operation).
- The value is changed.
- The status is changed (every access changes the status: see definition 12).
- The associated set of constraints is changed.

Based on these change operations, we can define an object local concurrency control method:

- If an operation tries to access an object, a check is performed if the new status satisfies the constraints (is the access allowed?). If this is not the case the access is denied.
- If the operation tries to change the value of the object, it is checked if the new value satisfies the constraints.All virtual objects which reference the object are checked as well. If the check fails, the access is denied.
- If the identity (the name) should be changed, it is checked if there are any constraints defined for the object or if there is a virtual object which references the object's value. If there are constraints or a reference, the access is denied.

- If the set of constraints should be changed we have two possibilities:
  (1) A new constraint is inserted: It is checked if B^o can be fulfilled. If this is not the case, the access is denied.
  If the new constraint contains a combination of values, it is checked if there exists a virtual object representing this combination. Otherwise a new virtual object is created. If the virtual object exists it is checked if the condition on the virtual object can be fulfilled.
  (2) An existing constraint is modified: We must check if the identification of the accessing operation allows the modification and if not we have to deny the access.

With a mechanism following the protocol described above we can ensure the correctness support of objects as well as the success support. Furthermore, an executing system can use the check during the modification (and insert) of constraints for application-oriented resolution of synchronization conflicts. This means that if an executing system tries to insert success supporting constraints before the execution of an operation and the access is denied, the executing system receives information about the conflict and not only an abort message of a failed transaction. If the application defines which activities to execute in case of this special event, it is tried to resolve the conflict instead of only using retry mechanisms. This can be used to implement, for example, a check/revalidate mechanism (Peinl et al. 1988).

Spheres – an abstraction

A system running the introduced process definitions (definition 7) sees primarily only avtivities and transitions instead of data objects. Because the activities in our system are transactional there are only state changes if an activity commits. Therefore, we need a mechanism which checks at least at the commit time of an activity,if the state change is allowable and if all objects are in an extended correctness supporting state.

From this point of view, definition 19 can be seen in a more abstracted way. An executing system of process definitions has to maintain a sphere of isolation which encloses an activity and its corresponding compensation activity to ensure the executability of the compensation activity. This means that the original activity defines system states which are valid for the successful execution of its compensation and which have to be guaranteed until the execution of the compensation or until we know that there never will be a compensation (a final state is reached). Activities of the same process definition executed later behave in the same way and therefore they join and modify the sphere of isolation. The result of a modification of the sets of allowed states must be a subset of the original allowed states and therefore we can say that the sphere of isolation ensuring the correctness of a computation is shrinking by executing activities and growing by executing compensation activities. This phenomenon is very similar to the 2-Phase-Locking schemes and we will discuss later which advantages our approach has.

A second sphere of isolation is the sphere for the successful executability (definition 19) of the "normal" computation. This sphere can be seen as an orthogonal mechanism to the sphere for correctness. In the extreme case the success-ensuring sphere guarantees after the commit of the first activity of a process definition that all other activities of the computation can be executed successfully. This means that after the first committed activity constraints on all objects are needed by the computation and these constraints are relaxed the more activities have been committed. In other words, the sphere of isolation is growing the more activities are executed successfully. This approach has some drawbacks. One is the need for "preclaiming"required objetcs which cannot be realized in general (see before). Another one is the unnecessary protection for activities which never have to be executed. A simple example is a conditional branch (if/case) where one or several paths never are followed.

Due to the drawbacks of the extreme case of a sphere of isolation for the succesful execution, it is useful to relax the approach. Instead of ensuring the executability of the whole execution a more dynamical method is introduced. When an activity accesses some objetcs required by another activity (later in the execution) the activity can leave constraints on the objects which guarantee a state where the later activity can be executed. With this approach we have no absolute guarantee that the computation can ever be finished successfully, but we have a mechanism to increase the probability. Furthermore, we can allow the dynamical change of the sphere of isolation including the possibility to add allowed states as well as mechanisms to disallow states which results in the possibility for a high degree of parallelism.

Aside from the arguments for ensuring the successful execution and parallelism we gain another very important feature with the isolation sphere. Since we have the possibility to change the allowable states in any way we want we get a basic mechanism to support cooperative work without violating the correctness of our executions. Moreover, we can allow the violation of our success supporting criterion in a controllable way (e.g. after negotiation) or by taking temporal aspects into account.

SUMMARY

In this paper we have introduced a very flexible method for concurrency control. It is a mechanism which supports – aside from correctness ensuring – the implementation of advanced execution models like cooperative executions or negotiating executions. The implementability and the performance of such a mechanism was not discussed, but we see the need to research for efficient algorithms to manage the constraints on objects and to be able to evaluate accesses efficiently. First steps in this direction have been made in case of numerical datatypes (e.g. O'Neil 1986) but there is a need to develop mechanisms for complex datatypes.

We will implement and evaluate this method in the prototypical implementation of a ConTract system called APRICOTS (Schwenkreis 1993) to show the feasibility of the mechanism. One of the major problems to realize this synchronization method for long running computation is the need for recoverable concurrency control information. Since a forward recovery mechanism is used we have to manage the constraints on objects as reliable as the objects themselves. Database vendors do not offer such a mechanism (e.g. recoverable locks outside transactions) because it is undecidable if a constraint without a corresponding active computation can be deleted or not. More precisely, they fear that the databases will be filled up with locks until nothing goes on . . .

REFERENCES

P.A. Bernstein, V. Hadzilacos, N. Goodman (1987) "Concurrency Control and Recovery in Database Systems", Addison Wesley, 1987

C. T. Davies Jr. (1990) "Data processing spheres of control" IBM Systems Journal 17,2 (1978) 179-198

Ahmed K. Elmagarmid ed. “Database Transaction Models for Advanced Applications”, Morgan Kaufmann Publishers, Inc., 1992

Hector Garcia-Molina (1983) “Using Semantic Knowledge for Transaction Processing in a Distributed Database” ACM Transactions on Database Systems 8,2 (1983) 186-213

Jim Gray , Andreas Reuter (1993) “Transaction Processing: Concepts and Techniques”, Morgan Kaufmann Publishers, Inc., 1993

Jim Gray (1981) “The Transaction Concept: Virtues and Limitations” Proc. of the 7th VLDB Conference (1981) 144-154

Patrick E. O’Neil (1986) “The Escrow Transactional Method” ACM Transactions on Database Systems 11,4 (1986) 405-430

Peter Peinl, Andreas Reuter, Harald Sammer (1988) “High Contention in a Stock Trading Database: A Case Study” Proc. of the ACM SIGMOD Intern. Conf. on Management of Data, (1988) 260-268

Jim Melton, Alan R. Simon (1993) “Understanding the new SQL: A Complete Guide”, Morgan Kaufman Publishers, Inc., 1993

Friedemann Schwenkreis (1993) “APRICOTS - A Prototype Implementation of a ConTract System: Management of the Control Flow and the Communication System” Proc. of the 12th Symposium on Reliable Distributed Systems (1993) 12-21

Helmut Waechter, Andreas Reuter (1992) “The ConTract Model” chapter 7 in (Elmagarmid 1992) 219-263