Temporally Faithful Execution of Business Transactions

Werner Obermair[1] and Michael Schrefl[2]

[1] Institut für Wirtschaftsinformatik
Universität Linz, Austria
obermair@dke.uni-linz.ac.at
[2] School of Computer and Information Science
University of South Australia, Australia
schrefl@cs.unisa.edu.au

Abstract. Serializability is a prominent correctness criterion for an interleaved execution of concurrent transactions. Serializability guarantees that the interleaved execution of concurrent transactions corresponds to *some* serial execution of the same transactions. Many important business applications, however, require the system to impose a partial serialization order between transactions pinned to a specific point in time and conventional transactions that attempt to commit before, at, or after that point i n time. This paper introduces *temporal faithfulness* as a new correctness criterion for such cases. Temporal faithfulness does not require real-time capabilities but ensures that the serialization order of a set of business transactions is not in conflict with precedence requirements between them. The paper also shows how a temporally faithful transaction scheduler can be built by extending proven scheduling techniques.

1 Introduction

Transaction processing as it is discussed widely in literature and as it is supported in many commercial systems follows a prominent principle: Concurrent transactions are executed in an interleaved manner and serializability is used as the correctness criterion for the interleaving. This means that an interleaved execution of concurrent transactions is considered correct if it produces the same output and has the same effect on the database state as *some* serial execution of the same transactions. O f course, not all serial executions produce the same output and have the same effect.

Important business applications are not supported appropriately if transaction processing follows this principle. Consider the following scenario: In a trading system, a trader intends to adjust the selling price of a product at 12:00 noon. The trader intuitively expects that sales transactions performed before 12:00 noon should be performed on the old selling price, and that sales transactions performed after 12:00 noon should be performed on the new selling price. In deciding whether a sales transaction i s performed before or after 12:00 noon, the trader considers relevant the point in time when the customer commits to

B. Wangler, L. Bergman (Eds.): CAiSE 2000, LNCS 1789, pp. 462–481, 2000.

a sale, i.e., when the customer decides to buy. This point in time corresponds to the "handshake" between the seller and buyer in a traditional sales transaction; in a computer-based trading sytem, this point in time corresponds to the attempt to commit the sales transaction. If the attempt to commit the sales transaction is placed before 12:00 noon, the sales transaction has to be performe d on the old selling price and, therefore, serialized before the adjusting transaction. If the attempt to commit the sales transcation is placed after 12:00 noon, the sales transaction has to be performed on the new selling price and, therefore, serialized after the adjusting transaction.

We refer to transactions as contained in the described scenario as *business transactions*. Business transactions require a combination of concepts that is not yet appropriately provided by current transaction execution models. From a user's perspective, the required concepts are:

First, business transactions behave as if executed instantaneously. Although they may be actually executed over some time period, the end user will assume that each business transaction is performed at a particular point in time.

Second, business transactions can be linked to wall-clock time. This point in time indicates when the transaction is performed from the perspective of the end user, regardless when it is actually executed internally.

Third, two types of business transactions can be distinguished. Those which are performed (from the user's point of view) at a specific wall-clock time and those which are performed at the wall-clock time of the "handshake", i.e., at the request to commit the business transaction.

Fourth, real-time capabilities are not required. It is sufficient that the effect of the execution of a set of business transactions is the same as if each transaction had been executed instanteneously at its specified wall-clock time.

How can this desired semantics of business transactions be achieved in a system that guarantees only that the interleaved execution of concurrent transactions corresponds to some serial execution? In our scenario, starting the adjusting transaction at 12:00 noon is not sufficient. The transaction execution may be delayed due to concurreny conflicts, and there may be sales transactions reading the old price but attempting to commit after 12:00 noon. A solution for this particular case would be to incorporate the price of a product as a time-stamped attribute (perhaps in a historical database). The price change could have been recorded proactively, and sales transactions could read the current selling price just before they attempt to commit. This solution, however, is not general since it may not be applied in cases in which the time critical update relies on a complex computation. For example, consider the case that the new product price should be determined through an analysis of the market as it is at 12:00 noon. The market at 12:00 noon cannot be anticipated and, thus, no proactive price change can be recorded. Also using some form of multi-version concurrency control will not be sufficient as such, since multi-version concurrency control is typically based on internal processing time rather than on real-world time. Business transactions are better supported by a system that provides a general mechanism for influencing the order in which transactions are serialized.

In this paper, we present such a general mechanism. Its key advantage is that it is based on standard concurrency control techniques and applicable to conventional database systems. As such it does neither rely on sophisticated, special concurrency control mechanisms (such as multi-version concurrency control) nor does it require specialized database systems (such as temporal or real-time).

We introduce a new category of transactions, so-called *pinned transactions*. A user who submits a pinned transaction gives the transaction an explicit timestamp. If a pinned transaction needs to be aborted internally during its processing (e.g., because of a deadlock), the transaction is automatically restarted with the same timestamp. In our example scenario, the transaction adjusting the selling price would be a pinned transaction with timestamp 12:00 noon. Also conventional transactions, we refer to them as *unpinned transactions*, receive timestamps. The timestamp of an unpinned transaction is not given by a user, but is the current reading of the system clock when the transaction attempts to commit. If an unpinned transaction needs to be aborted internally during its processing, it is the user who decides whether to restart or to dismiss the transaction. A user who decides to commit an unpinned transaction relies on the database state as it is at the attempt to commit. It cannot be assumed t hat the user decides to issue the attempt to commit again if the database state has changed after a restart. In our example scenario, sales transactions would be unpinned transactions receiving as timestamps the reading of the system clock when they attempt to commit.

The timestamps associated with transactions do not mean that the transactions are executed real-time at the indicated points in time. A less stringent criterion than real-time execution is sufficient for many business applications: *temporal faithfulness*. Timestamps are interpreted as precedence requirements and temporal faithfulness means that the serialization of transactions follows these precedence requirements. Although possibly laging behind real time, a system that behaves in a temporally faithf ul manner guarantees the expected serialization order. Reconsider the transaction adjusting the selling price of a product upon an analysis of the market as it is at 12:00 noon: If this pinned transaction is still running after 12:00 noon, unpinned sales transaction carrying a later time stamp but incorrectly relying on the old selling price are prevented from committing. Moreover, if such an uncommitted sales transaction keeps the adjusting transaction from committing (e.g., because of a lock conflict), th e sales transaction is aborted.

We first encountered the need for a temporally faithful execution of transactions when we developed an execution model for *rules on business policies* in [9]. Rules on business policies express how an external observer, e.g., a swift clerk, could monitor calendar time and the database state to recognize predefined situations and to react to them accordingly in order to meet a company's business policy. They are formulated according to the event-condition-action structure of rules in acti ve database systems [4] and can be classified into two categories, (1) rules that are executed asynchronously to user transactions and (2) rules that are executed "immediately" upon temporal events. The second rule cate-

gory is not supported appropriately by the execution models of existing active database systems. Existing active database systems allow to react upon temporal events only by starting independent transactions without any possibility to predict when the reaction will actually be committed to the database. A rule of the second category, however, relies on a temporally faithful execution of transactions where the execution of the rule is reflected by a pinned transaction. For details on rules on business policies and their execution semantics see [9].

The remainder of this paper is organized as follows: Section 2 introduces the concept of temporal faithfulness. Section 3 establishes a theory for a concurrent and temporally faithful execution of transactions. Section 4 outlines a scheduler that provides for a temporally faithful execution of transactions and that is based on strict two-phase locking. Section 5 concludes the paper by summarizing its main achievements and by outlining future work .

2 Temporal Faithfulness

In this section, we introduce the concept of temporal faithfulness. Section 2.1 starts with the presentation of the time model underlying our timestamping mechanism. Section 2.2 discusses the precedence requirements imposed by the timestamps assigned to transactions. Section 2.3 relates temporal faithfulness with concepts presented in the diverse literature.

2.1 Time Model

The timestamps assigned to pinned transactions bear application semantics. Timestamps have to be meaningful for users. Time as it is perceived by users may be of a coarser granularity than it is supported in some database system. In other words, the temporal unit that is not further decomposable from a user's perspective can often be of a coarser granularity than the temporal unit defined by two successive clock ticks of an internal clock of some target system. E.g., timestamps showing only minutes but not seconds are sufficient for many business applications. We consider time as a finite sequence of chronons. A chronon is the smallest temporal unit that is not further decomposable from a user's perspective. The timestamps that are assigned to transactions—either explicitly in the case of pinned transactions, or upon their attempt to commit in the case of unpinned transactions—are of the same granularity as chronons. This means that several transactions will share a common timestamp.

2.2 Precedence Requirements

Timestamps that are associated with transactions impose precedence requirements that induce a partial execution order. For the sake of simplicity, we assume an execution model in which transactions are executed serially. We will relax this assumption in Sect. 3.

If a transaction t has a lower timestamp than another transaction t', temporal faithfulness requires that t has to be executed before t'. Irrespective of chronon length, there may be several transactions that share a common timestamp. Even if chronons are extremely short, the timestamp given to a pinned transaction by an external user may coincide with the point in time at which an unpinned transaction requests to commit. Several pinned transactions with the same timestamp cannot be further distingu ished and, therefore, no precedence requirements can be identified among them. Correspondingly, also several unpinned transactions with the same timestamp can be executed in an arbitrary order. Pinned and unpinned transactions sharing a common timestamp, however, may be related in different ways:

1. *Head policy:* All pinned transactions are performed before all unpinned transactions. In this policy, the unpinned transactions are performed on a database state that reflects the updates of the pinned transactions. Scenarios can be drawn in which this execution semantics is favourable.

   Example 1. Consider a pinned transaction that updates selling prices at 12:00 noon upon market analysis. This pinned transaction is expected to be performed before any unpinned sales transaction attempting to commit during the chronon from 12:00 to 12:01 noon. The sales transactions should be performed on the new price.

2. *Tail policy:* All pinned transactions are performed after all unpinned transactions. In this policy, the pinned transactions are performed on a database state that reflects the updates of all unpinned transactions. Favourable scenarios can be drawn, too.

   Example 2. Consider a transaction pinned to 11:59 a.m. that publishes the selling prices and computes the morning sales figures. This pinned transaction is expected to be performed after all unpinned sales transactions attempting to commit during the chronon from 11:59 a.m. to 12:00 noon. The sales performed during the last chronon of the morning should be reflected in the sales figures.

3. *Don't care policy:* Pinned transactions and unpinned transactions are executed in arbitrary order. If a pinned transaction precedes some unpinned transactions with the same timestamp and succeeds some others, it is not predictable on which database states transactions are executed. We did not encounter any realistic application scenario that is best supported by the *don't care policy* and believe that this case is probably not relevant in practice.

We support the *head policy* and the *tail policy* by introducing two kinds of pinned transactions: transactions that are pinned to the begin of a chronon (*head transactions*) and transactions that are pinned to the end of a chronon (*tail transactions*). One may argue that a tail transaction t pinned to the end of chronon i (case 1) could be replaced by a head transaction t' pinned to the

begin of chronon $i + 1$ (case 2). This is not true for the following reason: In the first case, all head transactions pinned to the begin of chronon $i + 1$ work on a database state that reflects the changes performed by t, in the second case, there may be some head transactions pinned to the begin of chronon $i + 1$ that are executed before t' since transactions of the same kind sharing a common timestamp are performed in any order.

From the view point of a chronon i—lasting from clock-tick i to clock-tick $i+1$—there are transactions pinned to the begin of chronon i (*head-transactions*, H^i), unpinned transactions attempting to commit during i (*body-transactions*, B^i), and transactions pinned to the end of chronon i (*tail-transactions*, T^i).

2.3 Related Work

Timestamps that are associated with transactions have a long tradition in transaction processing: They are employed in timestamp ordering as a concurrency control mechanism or in deadlock prevention without risking livelocks. The commit time of transactions is often used as transaction time in rollback or bitemporal databases. Commit-time timestamping guarantees transaction-consistent pictures of past states of a database ([11]). In all those cases, however, the timestamps assigned to tra nsactions are generated by the database system and do not bear application semantics.

Georgakopoulos et al. [7] introduce transaction timestamps that bear application semantics. They call their timestamps "value-dates". By means of value-dates, Georgakopoulos et al. specify "succession dependencies". Like precedence requirements in our model, succession dependencies do not impose real-time constraints on the execution of transactions, but they influence the ordering of transactions. If a succession dependency is specified between two transactions, the two transacti ons are serialized in the requested order irrespective whether they conflict in their operations or not. This is an important difference to temporal faithfulness where a precedence requirement influences only the ordering of *conflicting* transactions. Disregarding other transactions, two transactions that do not conflict in their operations can be serialized in an arbitrary order in a temporally faithful setting. We will discuss this in detail in Sect. 3.

Finger and McBrien [6] introduce "perceivedly instantaneous transactions" for valid-time databases. For a perceivedly instantaneous transaction it is guaranteed that "current time" (usually referred to as `now`) remains constant during its execution. The point in time at which a transaction is submitted is taken as the transaction's value of `now`. Transactions that conflict in their operations are serialized according to their values of `now`, i.e., according to their subm ission points. Contrary to our approach, timestamps assigned to transactions do not bear application semantics in the approach of Finger and McBrien. Further, all transactions are timestamped in the same way. This is an important difference to our approach where usually a high number of unpinned transactions is serialized around a few pinned transactions.

The specification of temporal transaction dependencies has been discussed in the literature also without using timestamps. For example: Ngu [10] builds prece-

dence graphs that reflect temporal dependencies. Dayal et al. [5] use rules as they are provided by active database systems to specify the ordering of transactions. They do this by specifying rules that are triggered by transaction events and by exploiting the capabilities of coupling modes (cf. [3]). The bas ic limitation of these approaches is that all related transactions must be known in advance. Our model is by far more modular and general: A transaction can be pinned to a point in time without the need to consider all the transactions that potentially may be executed around the critical point in time.

3 Theory of Concurrent and Temporally Faithful Histories

A theory to analyze the concurrent temporally faithful execution of transactions can be formulated similarly to the classical serializability theory (cf. [1]). We will establish our theory through the following steps: First, we will define the conditions under which a serial history is temporally faithful. Then, we will recall the conditions under which two histories are equivalent, and we will define a history to be temporally faithfully serializable—and thus correct—if it is equivalent to a temporally faithful serial history. Finally, we will show how it can be tested whether a history is temporally faithfully serializable.

A history covers a set of chronons C and a set of transactions T where T contains all committed transactions $t_1, t_2, \ldots, t_n$ that are time-stamped with a chronon in C. History h indicates the order in which the operations of $t_1, t_2, \ldots, t_n$ are executed. T can be subdivided into disjoint subsets according to two dimensions: (1) For every chronon c ($c \in C$) there is a subset T_c containing the transactions time-stamped with c. (2) According to the transactio n categories, T can be subdivided into a set of head-transactions T^h, body-transactions T^b, and tail-transactions T^t. Combining the two dimensions, for every chronon c ($c \in C$) there is a set T_c^h containing the head-transactions that are pinned to the begin of c, a set T_c^b embracing the body-transactions that attempt to commit during c, and a set T_c^t containing the tail-transactions that are pinned to the end of c.

Definition 1. *A history is* temporally faithfully serial *if it is serial, if it observes timestamps, and if it observes priorities:*

1. *A history is* serial *if, for every two transactions t_i and t_j, either all operations of t_i are executed before all operations of t_j or vice versa. Thus, a serial history represents an execution in which there is no interleaving of the operations of different transactions.*
2. *A history* observes timestamps *if, for every two transactions $t_i \in T_{c_k}$ and $t_j \in T_{c_l}$ where $c_k < c_l$, t_i is executed before t_j. Thus, timestamp observation requires that transactions with different timestamps are executed in timestamp order.*
3. *A history* observes priorities *if (1) for every two transactions $t_i \in T_{c_k}^h$ and $t_j \in T_{c_k}^b \cup T_{c_k}^t$, t_i is executed before t_j and if (2) for every two transactions $t_i \in T_{c_k}^b$ and $t_j \in T_{c_k}^t$, t_i is executed before t_j. Thus, priority observation*

requires that head-transactions are executed before body- and tail-transactions and that body-transactions are executed before tail-transactions if they have the sam e timestamp.

Example 3. We examine histories over the set of chronons $C = \{c_1, c_2\}$ and the set of transactions $T = \{t_1, t_2, t_3, t_4, t_5\}$ with[1]

$t_1 = r_1[x] \rightarrow w_1[x]$
$t_2 = r_2[z] \rightarrow w_2[z]$
$t_3 = r_3[y] \rightarrow r_3[x] \rightarrow w_3[y]$
$t_4 = r_4[z] \rightarrow r_4[x] \rightarrow w_4[z]$
$t_5 = r_5[y] \rightarrow w_5[y]$

where $T^h_{c_1} = \{\}$, $T^b_{c_1} = \{t_1\}$, $T^t_{c_1} = \{\}$, $T^h_{c_2} = \{t_2\}$, $T^b_{c_2} = \{t_3, t_4\}$, and $T^t_{c_2} = \{t_5\}$.

Different timestamps and different priorities impose precedence requirements on the transactions contained in T. Figure 1 shows the imposed precedence requirements graphically. A circle depicts a transaction, an arrow from a transaction t_i to a transaction t_j indicates that t_i has to be performed before t_j. In our example: Timestamp observation requires for a temporally faithfully serial history that transaction t_1 is executed before transactions t_2, t_3, t_4, and t_5. Priority observation requires that head-transaction t_2 is executed before body-transactions t_3 and t_4 and before tail-transaction t_5 and that body-transactions t_3 and t_4 are executed before tail-transaction t_5.

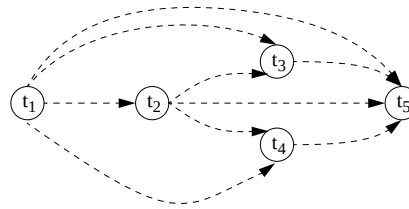

Fig.1. Precedence requirements imposed by timestamps and priorities

Definition 2. *Two histories h and h' are* equivalent *if (1) they cover the same set of transactions performing the same operations, and if (2) they order conflicting operations in the same way. The second condition requires that for any conflicting operations o_i belonging to transaction t_i and o_j belonging to transaction t_j, their execution order in h corresponds with their execution order in h'. (This definition follows the basic serializability theory, cf. [1]).*

Definition 3. *A history is* temporally faithfully serializable (TFSR) *if it is equivalent to a temporally faithfully serial history.*

[1] We denote the ordering of operations within a transaction or history by means of arrows ($\rightarrow$).

Whether a history is temporally faithfully serializable can be determined by checking an extended form of a serialization graph (SG), a so-called *temporally faithful serialization graph (TFSG)*, for cycles. Like an SG, a TFSG is a directed graph whose nodes are the transactions covered by the analyzed history. The edges of a TFSG, however, do not represent only precedence requirements imposed by conflicts, but also precedence requirements imposed by different timestamps and priorities. Throughout the rest of this paper, we refer to precedence requirements imposed by different timestamps or different priorities as *temporal precedence requirements*.

A TFSG is built in two steps:

1. An SG is built by introducing edges that represent precedence requirements imposed by conflicts. An SG for a history h contains an edge from transaction t_i to transaction t_j ($i \neq j$) if t_i issues an operation o_i that conflicts with an operation o_j of t_j and if o_i precedes o_j in h. An edge from t_i to t_j expresses that t_i has to precede t_j in a serial history equivalent to h. According to the classical Serializability Theorem, an equi valent serial history can be found—and thus a history is serializable—iff its serialization graph is acyclic (for the Serializability Theorem and a proof see [1]). An acyclic SG means that the precedence requirements imposed by conflicts do not contradict each other.

   Example 4. We continue the above example. Consider the histories:

   $$h_1 = r_5[y] \rightarrow w_5[y] \rightarrow r_2[z] \rightarrow r_3[y] \rightarrow r_3[x] \rightarrow w_2[z] \rightarrow r_4[z] \rightarrow w_3[y] \rightarrow r_1[x] \rightarrow r_4[x] \rightarrow w_1[x] \rightarrow w_4[z]$$
   $$h_2 = r_1[x] \rightarrow r_2[z] \rightarrow w_1[x] \rightarrow r_3[y] \rightarrow w_2[z] \rightarrow r_4[z] \rightarrow r_3[x] \rightarrow r_4[x] \rightarrow w_4[z] \rightarrow w_3[y] \rightarrow r_5[y] \rightarrow w_5[y]$$

   Figure 2 shows the SG for history h_1 in part (a) and the SG for history h_2 in part (b). The edges representing precedence requirements imposed by conflicts are shown as solid lines. Both graphs are acyclic, i.e., for both histories equivalent serial histories can be found.

2. The SG is extended to a TFSG by adding edges that represent temporal precedence requirements. Since a cyclic SG implies a cyclic TFSG, this second step is applied only if the SG of a history is acyclic, i.e., if the history is serializable. To capture different timestamps, for every pair of transactions (t_i, t_j) with $t_i \in T_{c_k}$, $t_j \in T_{c_l}$ and $c_k < c_l$, there is an edge from t_i to t_j. To capture different priorities, for every pair of transactions (t_i, t_j) with (1) $t_i \in T^h_{c_k}$ and $t_j \in T^b_{c_k} \cup T^t_{c_k}$, or with (2) $t_i \in T^b_{c_k}$ and $t_j \in T^t_{c_k}$, there is an edge from t_i to t_j. If a TFSG is built according to these rules, the TFSG contains also all edges implied by transitivity: If there is an edge from t_j to t_k and an edge from t_k to t_l, there is necessarily an edge from t_j to t_l. We draw edges implied by transitivity since they allow to efficiently decide whether a TFSG is acyclic (see below).

   Example 5. We continue the above examples: Figure 2 (c) shows the TFSG for h_1 and Fig. 2 (d) shows the TFSG for h_2. The edges representing temporal precedence requirements are shown as dashed lines.

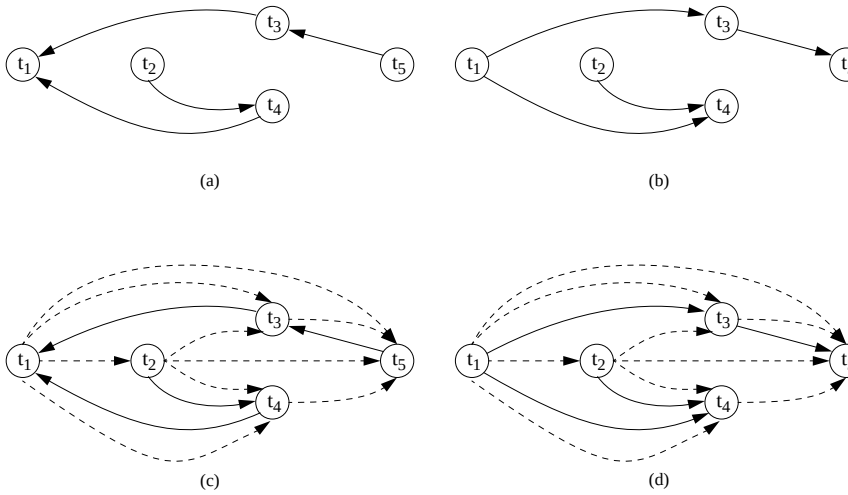

Fig.2. Building TFSGs (example)

A history is TFSR if there are no contradicting precedence requirements. Precedence requirements contradict, for example, if conflicts require transaction t_i to be executed before transaction t_j in a serial execution whereas timestamp observation requires t_j to be executed before t_i. Contradicting precedence requirements become visible in TFSGs as cycles. This can be summarized in a theorem closely related with the classical Serializability Theorem:

Theorem 1. *A history is TFSR iff its TFSG is acyclic.*

Proof. See Appendix.

Example 6. We continue the above examples: Figure 2 shows that the TFSG of h_1 contains cycles. History h_1 is not TFSR since the precedence requirements imposed by timestamps and priorities contradict the precedence requirements imposed by conflicts. The TFSG of h_2, however, is acyclic. History h_2 is TFSR since the precedence requirements imposed by timestamps and priorities do not contradict the precedence requirements imposed by conflicts.

Cycles can be detected efficiently in a TFSG if their characteristics are exploited:

Lemma 1. *Every cycle in a TFSG contains at least one edge representing a precendence requirement imposed by a conflict.*

Proof. See Appendix.

Cycles that can be detected efficiently are cycles of length two. In fact, it can be shown that the existence of a cycle of an arbitrary length in a TFSG of a serializable history implies the existence of a cycle of length two. This is captured by the following theorem:

Theorem 2. *A TFSG of a serializable history is acyclic if it does not contain a cycle of length two.*

Proof. See Appendix.

Corollary 1. *A serializable history is TFSR if the serialization of every pair of conflicting transactions (t_i, t_j) is in accordance with possible temporal precedence requirements between t_i and t_j.*

Proof. See Appendix.

This insight reduces the effort of building a TFSG. In order to decide whether a serializable history is TFSR, it is sufficient to introduce an edge representing a temporal precedence requirement only if the two involved transactions conflict. Temporal precedence requirements between transactions that do not conflict can be neglected. If no cycle (of length two) arises, the history is TFSR.

4 A Temporally Faithful Scheduler

This section presents a scheduler that provides a temporally faithful and of course serializable execution of concurrent transactions. First, we select strict two-phase locking as the basis for discussing a temporally faithful scheduler. Then, we present a temporally faithful scheduler built on strict two-phase locking. Finally, we sketch possible enhancements to the presented scheduler.

4.1 Rationale

Similar as serialization-graph testing (SG-testing) can be used in conventional systems to ensure serializability, TFSG-testing could be used to ensure temporal faithfulness. SG-testing is rarely used in conventional systems. It suffers from the considerable overhead of maintaining a graph and checking for cycles. Since TFSG-testing implies SG-testing, TFSG-testing suffers from the same problems and does not seem promising. One of the most prominent scheduling techniques, implemented by many commercial data base systems, is *strict two-phase locking* (strict 2PL). We select strict 2PL as the basis for our considerations since we want to discuss temporally faithful scheduling on a broad and well known basis.

Strict 2PL guarantees serializability of a produced history. An add-on is necessary to achieve temporal faithfulness. The add-on has to ensure that the serialization order imposed by the underlying 2PL scheduler does not contradict the temporal precedence requirements. As we have shown in Corollary 1, a serializable history is TFSR if the serialization of every pair of conflicting transactions (t_i, t_j) is in accordance with possible temporal precedence requirements between t_i and t_j. Thus, the add-on has to check only the serialization order of pairs of conflicting transactions.

In strict 2PL, conflicts occur only between two transactions that run concurrently. Only then, a lock conflict arises if the two transactions try to perform

conflicting operations. When a conventional strict 2PL scheduler encounters a lock conflict, the scheduler forces the requesting transaction (requester) to wait until the holding transaction (holder) releases its locks, which temporally coincides with the commit of the holding transaction. The scheduler serializes the requester after the holder. In a te mporally faithful setting, this may contradict the temporal precedence requirements between the involved transactions. A temporally faithful scheduler has to behave according to the principle *"abort holder if younger"*. If the holder of a lock has a higher timestamp or the same timestamp but a lower priority than the requester of an incompatible lock, the holder has to be aborted. Otherwise, the requester has to be blocked.

If two transactions do not run concurrently, an existing conflict between them does not become visible. A way to avoid that conflicts are missed is to delay the commit of transactions. A temporally faithful scheduler built on strict 2PL cannot grant the commit of a transaction t before it is sure that no lock conflict can arise upon which t would have to be aborted. No such lock conflict can arise after all transactions that have a lower timestamp than t or that have the same timestamp as t but a hi gher priority are committed.

Example 7. Consider a tail-transaction with timestamp 11:59 a.m. that publishes the selling prices and computes the morning sales figures. Further, consider a head-transaction with timestamp 12:00 noon that adjusts the selling prices of all items according to market analysis. Obviously, the two transactions conflict, and the tail-transaction has to be serialized before the head-transaction. Assume that the head-transaction attempts to commit already at 11:58 a.m. since it has been pre-scheduled (see below). If the c ommit is granted and the locks are removed, conflicts between the transactions do not become visible. The commit of the head-transaction has to be delayed until all transactions with a lower timestamp and all transactions with the same timestamp but a higher priority are committed. Only then, a lock conflict arises and the head-transaction can be aborted (and restarted).

In a system where a commit is not granted immediately, the point in time when the execution of a pinned transaction is started influences the system's behavior significantly. The start time determines whether a pinned transaction may have to wait for its commit rather long or whether other transactions may have to wait for the commit of the pinned transaction. Two extreme approaches in starting pinned transactions are:

1. *Full pre-scheduling:* A pinned transaction is started immediately when it is submitted. This approach may be inefficient if the transaction is submitted pro-actively long before its timestamp. Then, the transaction has to wait rather long for its commit and has to be aborted every time it runs into a lock conflict with a transaction carrying a lower timestamp.
2. *No pre-scheduling:* A pinned transaction is started only after all transactions that have to precede it are committed (i.e., the transaction is not "pre-scheduled"). This approach may be inefficient if the transaction performs

time-consuming operations. Then, other transactions will have to wait rather long for the commit of the pinned transaction.

Both approaches are not satisfactory for all application scenarios. Different application scenarios need different start times for pinned transactions. Therefore, we assume that the user who submits a pinned transaction specifies the point in time when the transaction has to be started. Depending on the nature of a transaction, the user may use the submission time, the timestamp of the transaction, or any time in between as the start time.

By forcing a transaction to wait upon its attempt to commit, some form of *two-phase commit* is introduced. In the first phase, the scheduler receives a commit request from a transaction that is ready to commit and registers the request for future handling. In the second phase, the scheduler actually grants the commit request and waits for the commit to be performed.

4.2 The TFSR Scheduler

For our presentation, we need a common understanding how transaction processing is done in a database system. Similar to Bernstein et al. [1], we suppose a modularized architecture in which a transaction manager, a scheduler, and a data manager contribute to transaction processing: The transaction manager (TM) performs any required preprocessing of transaction operations it receives from applications. It uses the services of the scheduler to schedule transaction operations and to commit or abort transactions. The scheduler provides its functionality by utilizing the services of a data manager (DM), which is responsible for recovery and cache management. In the following, we restrict our discussion to the scheduler, in particular, to those scheduling services in which a temporally faithful scheduler differs from a conventional strict 2PL scheduler. We do not further elaborate on the TM and the DM in the scope of this paper.

Like a conventional scheduler, the TFSR scheduler concurrently provides services for scheduling operations, for handling commit requests, and for handling abort requests. In handling abort requests, the TFSR scheduler does not differ from a conventional scheduler. In scheduling operations and in handling commit requests, however, the scheduler deviates from standard strategies. The scheduler cannot grant commit requests immediately and has to deal with lock conflicts in the realm of temporal precedence requ irements. Further, the scheduler requires a new transaction to be registered before its operations can be scheduled. A transaction's timestamp has to be accepted. The TFSR scheduler does not act only upon invocation of one of its services but also as time proceeds. The scheduler steps forward from chronon to chronon and grants pending and arriving commit requests of transactions.

In the following, we discuss how the TFSR scheduler registers transactions, how it schedules operations, how it handles commit requests, and how it grants the commit of transactions:

Registering Transactions. Before the scheduler may schedule any operation of a transaction, the transaction has to be registered with the scheduler. In the case of a pinned transaction, the scheduler has to check the transaction's timestamp. A pinned transaction could theoretically be time-stamped with c at every time. Then, however, a temporally faithful scheduler could never grant the commit of a transaction with a timestamp higher than c. It would never hold that all transactions with tim estamp c had been committed. We therefore allow pinned transactions to be scheduled only pro-actively. We refer to the current reading of the wall-clock time reduced to chronon granularity as WCT. In particular, the timestamp assigned to a head-transaction must be greater than the WCT and the timestamp assigned to a tail-transaction must be greater than or equal to the WCT.

Scheduling Operations. As motivated above, the TFSR scheduler acts according to the principle *"abort holder if younger"* if a lock conflict arises. The strategy of aborting and blocking transactions relies on transaction timestamps: The timestamp of a pinned transaction is known immediately when the transaction is registered. The timestamp of a body-transaction, however, is unknown before the transaction's commit request. The timestamp of a body-transaction that has not yet requested to commit is resolved dynamically to the WCT, i.e., to the lowest timestamp potentially assigned to the transaction.

Alternatively, it would be possible to block a requesting body-transaction irrespective of the WCT. The produced serialization order would still remain valid since the timestamp of the blocked body-transaction would increase as time proceeds. Such a blocking, however, usually cannot be accepted.

Example 8. Assume a head-transaction with timestamp 12:00 noon that adjusts selling prices. Further assume that the transaction has been pre-scheduled and is waiting for its commit. Let the WCT be 11:55 a.m. If now a sales transaction (body-transaction) runs into a lock conflict with the head-transaction, it cannot be accepted that the sales transaction is delayed for such a long time. Rather, the timestamp of the sales transaction is resolved to 11:55 a.m., the two timestamps are compared, and the head-transaction is aborted and automatically restarted.

When the timestamp of a body-transaction is resolved dynamically, a decision to force a pinned transaction to wait until a body-transaction has committed may become invalid as time proceeds. This means that lock tables and queues of blocked transactions have to be reconsidered as time proceeds (and the WCT changes).

Example 9. We continue the above example. Assume that the customer hesitates to commit the sales transaction. Let the WCT be 11:58 a.m. in the meantime. If now the restarted head-transaction runs into a lock conflict with the sales transaction again, the head-transaction is forced to wait. If the WCT increases to 12:00 noon, however, the decision to block the head-transaction becomes invalid. The sales transaction has to be aborted, and lock tables and queues have to be adjusted accordingly.

The presented strategy for resolving precedence requirements is only a supplement to existing strategies for detecting and dealing with deadlocks. Conventional deadlocks are outside the scope of our paper, they may still occur for transactions among which no temporal precedence requirements are defined (i.e., among transactions sharing a common timestamp and priority).

If a head- or tail-transaction needs to be aborted, the transaction is restarted automatically with the same timestamp. The transaction manager re-submits the transaction's operations for scheduling. If a body-transaction needs to be aborted, the transaction is *not* restarted automatically. It is the user who decides whether to restart or dismiss an aborted body-transaction.

Handling Commit Requests. When a transaction requests to commit, the transaction is not committed immediately, but is marked ready-to-commit. In the case of a body-transaction, the transaction additionally is time-stamped with the WCT. Transactions that are marked ready-to-commit are actually committed when the scheduler considers the corresponding chronon (see below).

Granting Commits. The scheduler continuously steps forward from chronon to chronon and grants pending and arriving commit requests of transactions. We refer to the chronon the scheduler currently considers as the scheduler's *current processing time (CPT)*. The scheduler's CPT may lag behind the WCT (CPT $\leq$ WCT). Figure 3 depicts the scheduler's state "granting commits" as a UML statechart diagram (cf. [2]). After its creation, the scheduler resides in this substate concurrently to other substates in which it registers transactions, schedules operations, and registers commit/abort requests. The diagram shows the steps the scheduler runs through after changing its CPT to chronon c_i:

1. *Grant commit requests of head-transactions:* In this state (named "head"), the assertion holds for a head-transaction t with timestamp c_i that all transactions that have to precede t have been committed. First, the scheduler commits head-transactions with timestamp c_i that have been marked ready-to-commit in the past. Then, the scheduler waits until all head-transactions with timestamp c_i have been committed. If a head-transaction with timestamp c_i is marked ready-to-c ommit in this phase, the commit is granted immediately.
2. *Grant commit requests of body-transactions:* In this state (named "body"), the assertion holds for a body-transaction t with timestamp c_i that all transactions that have to precede t have been committed. First, the scheduler commits body-transactions with timestamp c_i that have been marked ready-to-commit in the past. As long as the WCT corresponds to c_i, the scheduler commits a body-transaction that is marked ready-to-commit. After the WCT increases beyond c_i, no further body-transaction is time-stamped with c_i (but with a higher timestamp).
3. *Grant commit requests of tail-transactions:* In this state (named "tail"), the assertion holds for a tail-transaction t with timestamp c_i that all transactions that have to precede t have been committed. First, the scheduler commits

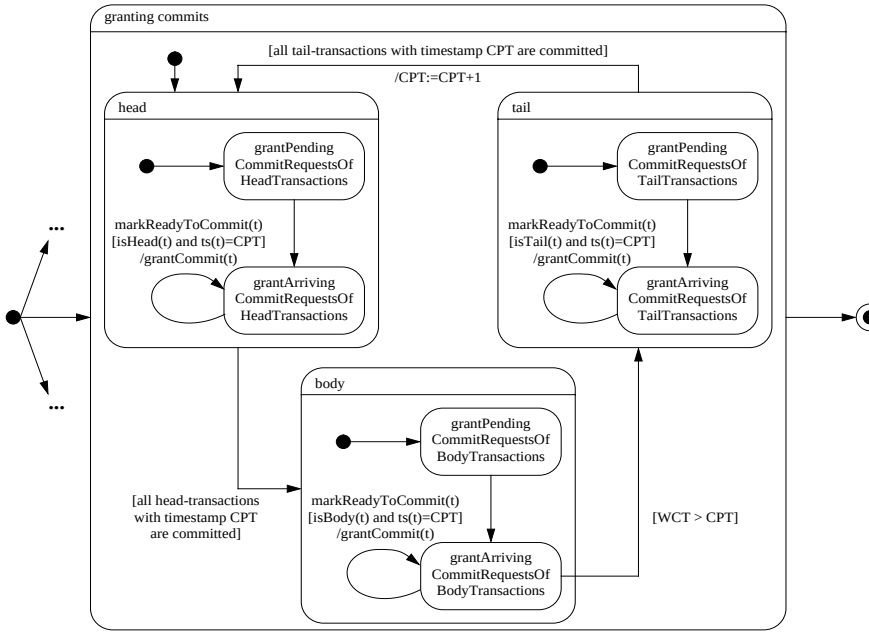

Fig.3. Granting commits (a substate of the scheduler)

tail-transactions with timestamp c_i that have been marked ready-to-commit in the past. Then, the scheduler waits until all tail-transactions with timestamp c_i have been committed. If a tail-transaction with timestamp c_i is marked ready-to-c ommit in this phase, the commit is granted immediately.

After running through these states, the scheduler has finished processing of chronon c_i and changes its CPT to c_{i+1}.

4.3 Advanced Scheduling Techniques

The performance of a temporally faithful scheduler can be increased if it has more knowledge at hand about what is going to be scheduled. In the following, we sketch advanced scheduling techniques that employ information on predeclared read-sets and write-sets of transactions, on the structure of pinned transactions, or on the expected durations of body-transactions.

– *Predeclared read-sets and write-sets of transactions:* The commit of a transaction can be granted if there is no conflicting transaction that has to be executed before. Without additional knowledge, every transaction has to be treated as a potentially conflicting one. All conflicts between a transaction t waiting for its commit and a transaction t' that has to be executed before t have become visible only after t' has committed. If no conflict arises, the

commit of t has been delayed unnecessarily. Conflicts can be detected earlier and, thus, long delays can be avoided if transactions preclaim all their required locks. This can be achieved by having each transaction predeclare its read-set and write-set. Predeclaring of read-sets and write-sets can be introduced only for pinned transactions or for all kinds of transactions.

- *Structure of pinned transactions:* In typical applications, pinned- and unpinned transactions more often run into read-write conflicts than into write-write conflicts. Remember the introductory example: Assume that the pinned transaction adjusting selling prices is waiting for the scheduler to grant its commit request. The pinned transaction has to be aborted and restarted every time a sales transaction with a lower timestamp tries to read price information that has been updated—and write-loc ked—by the pinned transaction. Restarts of pinned transactions could be significantly reduced if pinned transactions were clearly separated into a read-phase and a write-phase: When a pinned transaction is started, it enters its read-phase. The transaction obtains the necessary read-locks, reads the database objects it needs, and performs all its time-consuming computations. Then, the transaction waits until the scheduler's CPT corresponds to the transaction's timestamp. While the transaction is waiting, it holds only read-locks and, thus, is less likely to run into lock conflicts. Only when the CPT corresponds to the transaction's timestamp, the transaction enters its write-phase. The transaction obtains the necessary write-locks (or upgrades some of its read-locks) and actually performs write operations.
- *Duration of body-transactions:* The timestamp of a body-transaction is resolved dynamically to the WCT during its execution. If a body-transaction is long, its associated timestamp increases continuously. If the timestamp of a body-transaction increases beyond the timestamp of a blocked transaction, the body-transaction has to be aborted and the lock tables and the queues of blocked transactions have to be updated. This represents an overhead that can be reduced by associating with every body-t ransaction an estimation how long the execution of the transaction will probably take. In the presence of such an estimation, the timestamps assigned to body-transactions would be more realistic and would reduce overhead.

5 Conclusion

In this paper, we have presented an approach to execute business transactions in a temporally faithful manner. The main characteristics of our approach are:

- The approach is general. It relieves the designer from inventing a case-specific solution every time a particular precedence order should be enforced on the execution of transactions.
- The approach is modular. By means of pinned transactions, precedence requirements can be imposed without the need to consider all the unpinned transactions that potentially may be executed around the critical point in time.

– The approach is simple. A temporally faithful scheduler can be implemented by extending proven scheduling techniques in conventional database systems. It does neither rely on special concurrency control mechanisms nor does it require specialized database systems.

Currently, a prototype of a temporally faithful scheduler is being implemented within a master thesis. The prototype is built on top of the active object-oriented database system TriGS [8]. The reason to select an active object-oriented database system is that we consider active rules a well-suited concept for enhancing system capabilities without the need to introduce special-purpose mechanisms.

Appendix

Proof (Theorem 1). First, we prove that a history is TFSR if its TFSG is acyclic: We suppose that the TFSG of a history h is acyclic. An acyclic TFSG may be topologically sorted. Consider $t_1, t_2, \ldots, t_m$ as a topological sort of the TFSG of h, and let h_s be the history $t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_m$. History h_s is temporally faithfully serial since (1) it is serial, (2) it observes timestamps, and (3) it observes priorities (cf. Definition 1). Now we show that h is equivalent to h_s. If a transaction t_i performs an operation o_i and another transaction t_j performs a conflicting operation o_j where o_i precedes o_j in h, this is reflected by an edge from t_i to t_j in the TFSG of h. In any topological sort of the TFSG of h, t_i must appear before t_j. Since a history h_s is serial, all operations of t_i appear before any operation of t_j in h_s. Thus, we have proved that any two conflicting operatio ns are ordered in h in the same way as they are ordered in h_s (and that h and h_s are equivalent).

Second, we prove that a history is TFSR only if its TFSG is acyclic. We do this by showing that no cyclic TFSG can exist for a history that is TFSR: We suppose a history h that is TFSR. Since h is TFSR, there is a temporally faithfully serial history equivalent to h. We refer to this history as h_s. Now we suppose a cycle in the TFSG of h and let the cycle be $t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_k \rightarrow t_1$. An edge from t_i to t_j exists in the TFSG of h (1) if t_i performs an operation o_i and t_j performs a conflicting operation o_j where o_i precedes o_j in h, (2) if t_i has a lower timestamp than t_j, or (3) if t_i has the same timestamp as t_j but a higher priority. Since h_s is serial, an edge from t_i to t_j in the TFSG of h requires that t_i appears before t_j in h_s. The existence of the cycle implies that each of $t_1, t_2, \ldots, t_k$ appears before itself in h_s, w hich is a contradiction. So, no cycle can exist for a history that is TFSR.

Proof (Lemma 1). In our setting, time is linear. Transaction timestamps impose a linear order on transactions with different timestamps, while priorities impose a linear order on transactions with the same timestamp. Thus, timestamp observation and priority observation impose acyclic precedence requirements. At least one edge representing a precedence requirement imposed by a conflict is necessary to form a cycle in a TFSG.

Proof (Theorem 2). We prove that the existence of a cycle in a TFSG of a serializable history implies the existence of a cycle of length two. We use the following notational conventions: With $t_i \rightarrow t_j$ we denote an edge from t_i to t_j representing a precedence requirement imposed by a conflict. With $t_i \Rightarrow t_j$ we denote an edge from t_i to t_j representing a temporal precedence requirement. With $t_i \gg t_j$ we denote an arbitrary edge from t_i to t_j.

We assume a cycle involving the transactions $t_1, t_2, \ldots, t_n$ $(n > 2)$. The cycle contains at least one edge representing a temporal precedence requirement. This is true since the TFSG of a serializable history is built on an acyclic SG, and thus at least one edge representing a temporal precedence requirement is necessary to form a cycle. We refer to one of these edges as $t_n \Rightarrow t_1$ and let the cycle be $t_n \Rightarrow t_1 \gg t_2 \gg \ldots \gg t_{n-1} \gg t_n$.

Now, we analyze the precedence requirements among the nodes contained in the cycle. We consider the triple (t_n, t_i, t_{i+1}), where i initially is 1 and increases with each iteration by 1. Before the i-th iteration, $t_n \Rightarrow t_i$ holds. After the i-th iteration, either we have shown the existence of $t_n \Rightarrow t_{i+1}$ in the TFSG or we have detected a cycle of length two and stop analyzing. Depending on the kind of precedence requirement between t_i and t_{i+1}, the pr ecedence requirements among the triple of transactions may follow only one of two alternative patterns:

1. $t_n \Rightarrow t_i \Rightarrow t_{i+1}$: In this case, also edge $t_n \Rightarrow t_{i+1}$ exists in the TFSG.
2. $t_n \Rightarrow t_i \rightarrow t_{i+1}$: In this case, edge $t_n \Rightarrow t_i$ indicates that t_n either has a lower timestamp than t_i or the same timestamp but a higher priority. This means that t_{i+1} cannot be temporally independent from both, t_n and t_i. One of the following temporal precedence requirements must hold:
   (a) t_{i+1} succeeds t_i: Then, edges $t_i \Rightarrow t_{i+1}$ and $t_n \Rightarrow t_{i+1}$ exist in the TFSG.
   (b) t_{i+1} succeeds t_n and is temporally independent from t_i: Then, edge $t_n \Rightarrow t_{i+1}$ exists in the TFSG.
   (c) t_{i+1} succeeds t_n and precedes t_i: Then, edge $t_{i+1} \Rightarrow t_i$ exists in the TFSG, and we detect the cycle of length two $t_{i+1} \Rightarrow t_i \rightarrow t_{i+1}$.
   (d) t_{i+1} precedes t_i and is temporally independent from t_n: Then, edge $t_{i+1} \Rightarrow t_i$ exists in the TFSG, and we detect the cycle of length two $t_{i+1} \Rightarrow t_i \rightarrow t_{i+1}$.
   (e) t_{i+1} precedes t_n: Then, edges $t_{i+1} \Rightarrow t_n$ and $t_{i+1} \Rightarrow t_i$ exist in the TFSG and we detect the cycle of length two $t_{i+1} \Rightarrow t_i \rightarrow t_{i+1}$.

We show that we necessarily detect a cycle of length two after at most $n-1$ iterations: The analyzed cycle contains at least one edge representing a precedence requirement imposed by a conflict (see Lemma 1). This means that we find pattern 2 at least once. Now suppose that condition (a) or (b) would hold every time we find pattern 2. Then, after $n-1$ iterations, the TFSG would contain an edge $t_n \Rightarrow t_n$, which is clearly never the case. Thus, condition (c), (d), or (e) must hold at least o nce when we find pattern 2. But then, we detect a cycle of length two. We have shown that if a cycle exists in a TFSG built on an acyclic SG, then there is also a cycle of length two.

Proof (Corollary 1). Every cycle of length two contains an edge representing a precedence requirement imposed by a conflict (cf. Lemma 1). Thus, when a TFSG is checked for cycles, only pairs of conflicting transactions have to be considered. A history is TFSR and its TFSG is acyclic if no pair of conflicting transactions (t_i, t_j) can be found where conflicts require t_i to precede t_j while at the same time different timestamps or priorities require t_j to precede t_i.

References

1. P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.
2. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language—User Guide.* Addison Wesley, 1999.
3. A.P. Buchmann, J. Zimmermann, J.A. Blakeley, and D.L. Wells. Building an Integrated Active OODBMS: Requirements, Architecture, and Design Decisions. In *Proc. of the 11th Intl. Conf. on Data Engineering (ICDE)*, 1995.
4. U. Dayal. Active Database Management Systems. In *Proc. of the 3rd Intl. Conf. on Data and Knowledge Bases*, pages 150–167, June 1988.
5. U. Dayal, U. Hsu, and R. Ladin. Organizing Long-running Activities with Triggers and Transactions. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 1990.
6. M. Finger and P. McBrien. Concurrency Control for Perceivedly Instantaneous Transactions in Valid-Time Databases. In *Proc. of the 4th Intl. Workshop on Temporal Representation and Reasoning.* IEEE Comp. Soc. Press, 1997.
7. D. Georgakopoulos, M. Rusinkiewicz, and W. Litwin. Chronological Scheduling of Transactions with Temporal Dependencies. *VLDB Journal*, 3(3), 1994.
8. G. Kappel, S. Rausch-Schott, and W. Retschitzegger. A Tour on the TriGS Active Database System—Architecture and Implementation. In *Proc. of the ACM Symposium on Applied Computing, Atlanta, Georgia*, 1998.
9. P. Lang, W. Obermair, and M. Schrefl. Modeling Business Rules with Situation/Activation Diagrams. In A. Gray and P. Larson, editors, *Proc. of the 13th Intl. Conf. on Data Engineering (ICDE)*, pages 455–464. IEEE Computer Society Press, April 1997.
10. A.H.H. Ngu. Specification and Verification of Temporal Relationships in Transaction Modeling. *Information Systems*, 15(2):5–42, March 1990.
11. B. Salzberg. Timestamping After Commit. In *Proc. of the 3rd Intl. Conf. on Parallel and Distributed Information Systems, Austin, Texas*, 1994.