ELECTRONIC **WORKSHOPS IN COMPUTING**

Series edited by Professor C.J. van Rijsbergen

B Novikov, University of St Petersburg, Russia and J W Schmidt, University of Hamburg, Germany. (Eds)

Advances in Databases and Information Systems, Moscow 1996

Proceedings of the International Workshop on Advances in Databases and Information Systems (ADBIS '96). Moscow, 10-13September 1996

A Transaction Model for Handling Composite Events

G. Kappel, S. Rausch-Schott, W. Retschitzegger and M. Sakkinen

Published in collaboration with the
British Computer Society

BCS

Springer

A Transaction Model For Handling Composite Events

G. Kappel S. Rausch-Schott W. Retschitzegger M. Sakkinen*
Department of Computer Science
University of Linz, A-4040 Linz, Austria
email: {gerti,stefan,werner,markku}@ifs.uni-linz.ac.at

Abstract

Rule-based (re)active systems are a commonly accepted solution in the area of nonstandard applications in order to express an event-driven and constraint-driven system environment. Several attempts have been made to integrate active concepts into object-oriented databases and to extend active knowledge models to gain more and more expressive power and flexibility. Unfortunately, execution models of active systems do not fully exploit all the advantages of the provided knowledge models. Among the most challenging research problems is the development of a transaction model in the presence of so called composite events. Our approach of multi-parent subtransactions tries to fill this gap. Multi-parent subtransactions extend the well-known nested transaction model by allowing multiple transactions to start a subtransaction in cooperation. In particular, this paper discusses the impacts of our extensions on the locking protocol of the original nested transaction model.

Keywords and Phrases. Active object-oriented database systems, nested transactions, composite events, multi-parent subtransactions, inheritance of locks

1 Introduction

Research on active databases has been driven by a need for supporting (re-)active database functionality important for a number of non-traditional applications such as computer integrated manufacturing (CIM) and workflow management [1, 7, 8, 16]. Active database capabilities are now finding their way into many of the most popular commercial database management systems. In contrast to conventional database management systems, active database management systems perform certain operations automatically, either in response to certain events or in response to conditions being satisfied. Therefore, these systems are often called *reactive systems*. This reaction is based on knowledge stored inside the database system, determining when and how to react. In almost all active systems the representation of knowledge is based on the Event/Condition/Action paradigm (ECA paradigm). Hence, the database system is able to monitor the situation represented by an event and by one or more conditions and to execute the corresponding actions when the event occurs and the conditions evaluate to true [2].

Research efforts have been put to a great extent into the development of knowledge models especially to increase the expressive power of event specification languages without considering the consequences for the execution models. Among the most challenging research problems is the development of a transaction model in the presence of so called composite events, i.e., events representing a combination of other events [3, 4]. The nested transaction model has been identified to be an appropriate transaction model for AOODBSs. As will be seen in Section 3, the existing nested transaction model is not powerful enough to support a subtransaction mode with respect to multiple event signaling transactions. Thus, that transaction model is extended by introducing multi-parent subtransactions. The model of multi-parent subtransactions is based on a simple extension of the nested transaction model [12, 19]. It provides a means to naturally map the semantics of composite events onto appropriate transaction modes, i.e., parallel or sequential and dependent or independent transaction executions. The transactions signaling events which are part of a composite event are allowed to cooperate in starting a common subtransaction in which the rule, i.e., its condition

*On leave from the Dep. of Computer Science and Information Systems, University of Jyväskylä.

and/or its action get executed. The major prerequisite for this is that we allow for an explicit specification of transaction modes for each event part of a composite event within the context of a rule. In this paper, special emphasis is given on the locking protocol in the course of the extended model. In particular, lock inheritance is treated to ensure consistency for rules triggered not only by simple events but also by composite events.

The remainder of this paper is organized as follows: Section 2 identifies the problems arising from nested transaction models in the presence of composite events. To tackle these problems, Section 3 introduces the notion of multi-parent subtransactions by extending the nested transaction model and in particular discusses the impacts of our extensions concerning the inheritance of locks. Section 4 illustrates, on the basis of an example, how the new model is applied for active object-oriented database applications. Finally, in Section 5 a comparison to related approaches and a discussion of some open problems as well as ongoing research issues are given.

2 Events and Rules in the Context of Transactions

After motivating the use of nested transactions for simple events, we discuss the special requirements composite events pose on an underlying transaction model by means of an example. Note, that in the realm of this work a simple event denotes a (simple) message event. Any message sent to an object or an object class qualifies as a message event. Incorporating other kinds of simple events such as time events and external events is part of ongoing research.

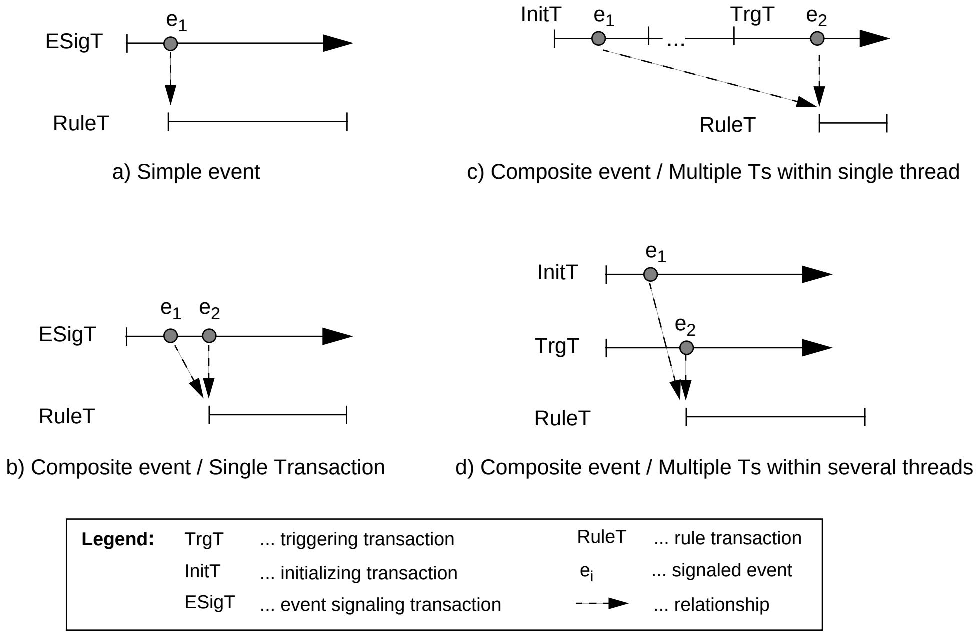
2.1 Nested Transactions for Simple Events

Most of the execution models of existing active object-oriented database systems are based on the nested transaction model [3, 4, 6, 8]. Note that in the following, if not stated otherwise, the term nested transactions stands for closed nested transactions [11]. The nested transaction model is particularly appropriate for active systems due to the following reasons [13, 20]: First, its structure accommodates nicely the hierarchical compositions of execution units as it is the case in active systems. The relationship(s) between *event signaling transaction*, i.e., the transaction in which an event is raised, and one or more *rule transactions*, i.e., the transactions wherein its condition and/or action are processed, can naturally be mapped to the relationship(s) between *parent transactions* and *child transactions* within a nested transaction model. Note, for ease of explanation it is assumed that condition and action are processed as atomic unit within a single rule transaction. For a discussion of condition and action being processed in different transactions we refer to [17], Second, nested transactions preserve serializability and top-level atomicity while allowing for a decomposition of a “unit of work” into subtasks as a prerequisite for intra-transaction parallelism. This is especially useful for active systems since rules often realize add-on functionality which may be executed in parallel to the event signaling transaction as well as to each other [23]. And third, comparing to traditional “flat” transaction models, finer-grained control over concurrency and recovery is provided [12], allowing to rollback rule transactions independently from event signaling transactions.

The different variations of nested transaction models used in active systems work well with simple events. A simple event triggering a rule establishes a one-to-one relationship between the event signaling transaction and the rule transaction processing the corresponding condition and/or action of the rule (Figure 1.a). Note, that in this paper we assume that an event triggers only a single rule. For a relaxation of this assumption we refer to [17]. The transaction mode for this one-to-one relationship in existing active systems is defined by means of coupling modes [3, 7]. A coupling mode in most systems allows to specify two possible transaction modes for a specific rule transaction: The rule transaction may be executed either in a subtransaction of the event signaling transaction which is executed serial to the event signaling transaction, or in an independent top-level transaction.

2.2 Nested Transactions for Composite Events

Simple events by themselves are not powerful enough to model complex real-world situations which have to be monitored by the active system [4]. Therefore, many active systems support the definition of *composite events*. They allow to construct composite events out of simple events and/or again other composite events by using different kinds of logical operators such as conjunction (AND), disjunction (OR), and sequence (;) [4, 9]. A composite event consists of so called *component events*. The event starting the detection of a composite event is called *initializing event*, and the event terminating the detection is called *terminating event* or *triggering event*. The transaction(s) wherein these component events occur are analogously called *event signaling transactions*, *initializing transaction* and *triggering transaction*. Following this terminology, both, an initializing transaction and a triggering transaction are special kinds of an event signaling transaction.


ESigT
e1
RuleT
InitT
e1
...
TrgT
e2
RuleT
ESigT
e1
e2
RuleT
InitT
e1
TrgT
e2
RuleT
Legend:
TrgT ... triggering transaction
InitT ... initializing transaction
ESigT ... event signaling transaction
RuleT ... rule transaction
ei ... signaled event
... relationship

a) Simple event

c) Composite event / Multiple Ts within single thread

b) Composite event / Single Transaction

d) Composite event / Multiple Ts within several threads

Figure 1: Origins of Events

With composite events more than one component event have to occur to trigger a single rule. Due to the origin of these component events three different cases can be distinguished: All component events occur in a single transaction (Figure 1.b), or some of them occur in different transactions. Concerning the latter case, there are again two possibilities: Either the event signaling transactions are sequentially processed within a single thread of control (Figure 1.c), or they are executed within different threads of control (Figure 1.d). Any of these cases results in a many-to-one relationship between event signaling transactions and a rule transaction. In existing active systems, this many-to-one relationship is handled in several different ways.

First, there are systems which do not allow component events of a specific composite event to occur in different transactions [4]. Second, there are systems allowing component events to occur in different transactions. However, in some of these systems the transaction modes which are responsible for specifying the semantics of the relationship(s) between triggering event(s), condition, and action refer only to the triggering transaction [9]. Third, there are systems which allow to specify transaction modes that refer to all event signaling transactions, but do not allow to specify a subtransaction mode [3]. The following example demonstrates that neither of these alternatives is appropriate to capture the semantics of real world situations.

Assume that a travel agency has the policy to reward regular customers by means of the following special offer: If a regular customer books both, a flight and a hotel room at the flight destination, a rental car may be reserved at a very special rate for one week. This policy can be best expressed by a rule which is triggered by a composite event as follows (note, we use a simplified notation borrowed from our underlying active object-oriented database system called TriGS, Triggersystem for GemStone [14, 15]):

```
DEFINE RULE reserveRentalCar AS
ON e1:(PRE(Hotel, reserve:roomRequest for:interval
              customer:aCust) AND
   e2: PRE(Airline, book:destination for:interval
              customer:aCust)) DO
IF checkCustomer /* regular customer &
              accepts cheap rental car offer */ THEN
EXECUTE rentalOffice reserve:compact for:e1.interval
              customer:aCust
END RULE reserveRentalCar.
```

The event part (`ON ...`) of the rule named `reserveRentalCar`, which is called *event selector* in TriGS, defines a conjunction (`AND`) of two simple message events. The first component event occurs each time after the message `reserve: for: customer:` has been sent to any object of class `Hotel`, but before (keyword `PRE`) the corresponding method has been performed. Analogously, the second component event occurs each time after the request to book a flight (message `book:for:customer:`) has been sent to any object of class `Airline`. The fact that both events are defined to be PRE message events allows to process condition and action of the rule without having to wait for hotel reservation and airline reservation. The condition part (`IF...THEN`) checks whether it is a regular customer, and whether (s)he accepts the offer for a cheap rental car. If so, i.e., if the condition evaluates to true, the action part is executed reserving a compact car for the specified interval (message `reserve:for:customer:` sent to `rentalOffice`). Note, that component events of a composite event are denoted by arbitrary names, cf `e1` and `e2` in our example. Accordingly, the interval used in the action part of the rule is referenced by the path expression `e1.interval` denoting the corresponding parameter within the first component event of the event selector.

In the context of this example, the transactional support in existing active systems is not appropriate. First, it is likely that the component events of our rule are signaled from within different transactions which may even be executed in different threads (cf. Figures 1.c and 1.d). This contradicts the first approach, which restricts component events to be signaled from within a single transaction only (cf. Figure 1.b). Second, one would like to express the fact that the failure of either the flight reservation or the room reservation may cause the car reservation to be canceled. This contradicts the second approach, which considers the transaction mode of a single triggering transaction only. However, this is not appropriate, since, due to the semantics of the conjunctive event operator, there is no deterministic triggering transaction and thus transaction modes cannot be assigned to a distinct transaction. Third, for the desired semantics within the example a subtransaction mode is appropriate. This is not only because the car reservation transaction is in a sense dependent on both event signaling transactions but in a way it is also required to access data which is shared with these transactions in a more cooperative manner than is possible between flat top-level transactions.

What is actually required is that the rule transaction, on the one hand, can be made dependent on both event signaling transactions and, on the other hand, is able to cooperate with them when accessing shared data. The next section suggests a solution for this problem.

3 Multi-Parent Subtransactions

In the context of composite events, where component events are signaled within different transactions, transaction modes should not only be applied to the triggering transaction but also to some or all of the other event signaling transactions. Our approach goes a step further by allowing the specification of distinct transaction modes for each component event in the context of a single rule, which at run time are applied to the corresponding relationships between event signaling transactions and rule transaction.

As soon as for different component events raised in different event signaling transactions a *subtransaction mode* is specified, within which the rule transaction gets executed, the nested transaction model is too restrictive. This is due to the fact that the nested transaction model does not provide any concept to capture several event signaling transactions related to a single rule transaction. In order to be able to express the semantics of several event signaling transactions, it is natural to extend the concept of a single parent transaction of the nested transaction model to multiple parents.

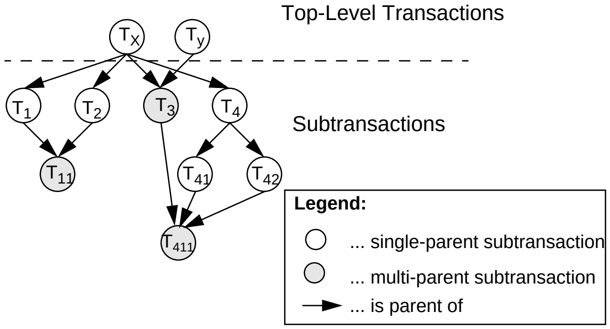


Figure 2: Multi-Parent Subtransactions

Therefore, in this section we introduce our approach of *multi-parent subtransactions* based upon the nested transaction model as defined by [12]. To be more specific, multi-parent subtransactions are applicable for the scenarios illustrated in Figure 1.c and 1.d. In Subsection 3.1 the basic properties of the extended transaction model, such as transaction structure, parallelism, dependencies and ACID properties are described. A detailed discussion of the locking protocol necessary to enforce these ACID properties is given in Subsection 3.2. In Subsection 3.3, the proposed locking protocol is illustrated by means of an abstract example.

3.1 Properties of Multi-Parent Subtransactions

Due to space limitations we will not describe the full model of nested transactions but rather our extensions and its impacts on the properties of the original model. Note that the term multiple parents (or multi-parents) in the following refers to *direct* parents of a subtransaction only, unless stated otherwise.

Transaction Structure. In the original nested transaction model, a nested transaction consists of a top-level transaction which may be decomposed into several subtransactions. Each subtransaction has exactly *one* parent transaction and in turn may span further subtransactions. Thus a so-called *transaction tree* is built. In contrast, our model allows every subtransaction to have an arbitrary number of direct parent transactions. Consequently, the structure of a nested transaction no longer resembles a tree, but a directed acyclic graph (Figure 2). At the very top level there may be more than one transaction, i.e., there is no single root transaction. Of course, more than one of such nested transaction graphs may be operational in parallel.

Transaction Parallelism. [12] allows subtransactions to be executed in parallel to each other as well as to their parent transaction. Since we support multiple parents, a subtransaction in our model may be executed in parallel to *some or all* of its parent transactions. In case that a subtransaction is defined as parallel to one or more parent transactions, each parent transaction has to be synchronized with the subtransaction at the latest when the parent is in a ready-to-commit state.

Transaction Dependency. Transaction dependencies are constraints over significant transaction events such as commit and abort [1]. According to the nested transaction model described in [12], the multi-parent subtransaction model specifies two types of dependencies: First, it requires that every parent transaction is *commit-dependent* on all of its childs. According to the definition in [5] this means that the parent transaction must not commit prior to the commit of those child transactions which are able to commit. Since in this case the commit of a parent includes all modifications done within the child transaction, consistency is ensured. Without the commit-dependency property, some update of a certain object within the parent transaction might get lost due to an update of the same object within the child transaction. Concerning child transactions which are intended to be aborted, their aborts need not be prior to the commit of the parent. For practical reasons, however, we assume the more general case that the commit of the parent transaction has to wait for the *termination* of all child transactions.

Second, besides commit dependency *abort dependency* is extended in order to cover multiple parents in that the child may be defined to be abort-dependent on *all* of its parent transactions. Unlike the commit-dependency, a parent may abort even if there are active abort-dependent child transactions. In this case, all active child transactions are notified by means of an abort signal.

For other kinds of dependencies between transactions which have been shown to be useful to support active systems we refer to [3].

ACID Properties. The ACID properties of our model conform to a great extent to the transaction model of [12]. This means that top-level transactions behave like a flat transaction having all ACID properties. To avoid premature commit violating isolation for top-level transactions, upward inheritance of locks has been introduced by [19]. For subtransactions intra-transaction parallelism is increased by relaxing *isolation* and *consistency* by means of controlled downward inheritance of locks (for upward and downward inheritance of locks we refer to Subsection 3.2). Note that this is in contrast to [19], where subtransactions still are fully isolated from other transactions inside the parent transaction. *Durability* in our model is slightly redefined in the way that committed changes of subtransactions are durable only after all top-level transactions within their ancestors have committed.

3.2 Locking Protocol for Multi-Parent Subtransactions

Both, upward inheritance of locks introduced in [19] as well as controlled downward inheritance of locks introduced in [12] have to be extended in order to cope with multi-parent subtransactions. The concept of upward inheritance assures that, upon commit of a subtransaction, changes are made visible to its direct parent transactions, only. By means of controlled downward inheritance parent transactions may offer locks they hold to direct and indirect subtransactions explicitly, thus ensuring that subtransactions can access objects which have already been accessed by the parent. In this way, safe parent/child cooperation on data structures to be read or written in a shared manner is achieved. Note, that in the original model of [19] which does not allow parent/child parallelism locks held by parent transactions are implicitly inherited to child transactions. Thus, child transactions by definition may access all objects which may be accessed by their parents.

Upward Inheritance. Upward inheritance has been defined by [12] as follows: As soon as a subtransaction commits, the direct parent transaction inherits all locks (held or retained) and retains these locks. [12] defines a *retained lock* as a place holder for a lock. The difference to holding a lock is that the *retainer* of a lock, i.e., the transaction that retains the lock, does not have the right to access the locked object. For (direct and indirect) parent transactions as well as transactions outside the hierarchy of the retainer, the retained lock resembles a held lock of the same mode, thus prohibiting to acquire a lock in a conflicting mode. Descendants of the retainer, on the contrary, are able to acquire conflicting locks.

In a first step upward inheritance as defined by [12] can be simply extended in the way that all parents inherit the subtransaction's locks and retain them. Descendents, siblings within the transaction graph which retain the same lock, and their descendents may acquire a lock in a mode conflicting to the retained mode (if no other transaction within the transaction graph *holds* a conflicting lock, of course). Moreover, in the context of multiple parents, the retained lock assures that only one of these parents, i.e., the first that tries to acquire the lock, gets the lock. Thus, objects locked in subtransactions remain isolated to the outside until every (direct and indirect) parent has successfully committed and, in this way, released the locks (cf. visibility rule and commit rule in [11]).

A major goal of any locking protocol is to keep the number of locks as small as possible in order to increase concurrency. The simple extension of inheriting all locks to the parents is not practical in case of a complex transaction graph, since there are situations where it is not necessary to inherit all locks to the parent. Inheriting all locks would result in an extensive distribution of locks to all transactions having a direct or indirect subtransaction in common. The criterion for the decision of which locks to be inherited upwards is the origin of a lock. There are three possible origins for locks within a subtransaction: First, the subtransaction may have acquired the lock by itself. Second, it may have inherited locks upwards from other subtransactions. And third, as is described below, locks may be inherited downward from its parent transactions.

Recalling the purpose of upward inheritance, it is sufficient to inherit only "new" locks, i.e., locks which have not been inherited downward. These are locks corresponding to the first two origins.

Figure 3 shows three different locks originating from different transactions. Note that, without loss of generality, a transaction model with a strict two-phase locking protocol and two different lock modes X, i.e., an exclusive lock, and S, i.e., a shared lock, is assumed in this example. In order to visualize the implications of lock inheritance, the figure also shows the sphere of the locks for each object. According to [12], the *X-sphere* (*S-sphere*) of a lock defines

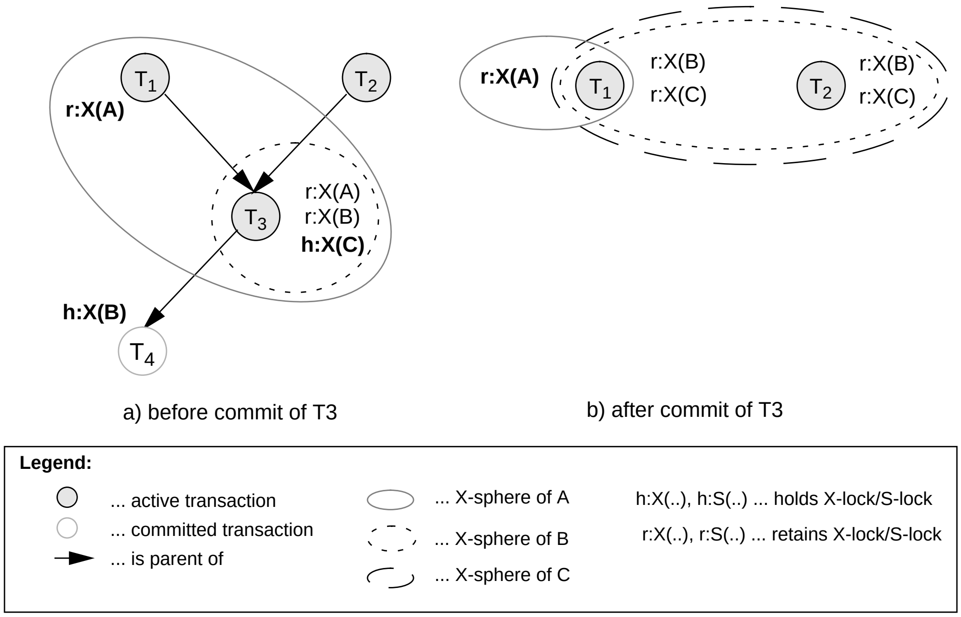


Figure 3: Lock Distribution Across Transactions

the set of transactions that can potentially lock a certain object in X-mode (S-mode). From the viewpoint of T3, the retained X-Lock on object A, r:X(A), is inherited from the parent T1, r:X(B) was inherited upwards from the child T4 which has already committed, and h:X(C) is acquired by T3 itself (Fig. 3a). Upon commit of T3 only X(B) and X(C) are inherited upwards to the parents T1 and T2. X(A) is not inherited upwards to T2, since it has not been used by T3 in order to change object A (Fig. 3.b). In this way, excessive lock inheritance is avoided.

Controlled Downward Inheritance. Concerning *controlled downward inheritance*, any parent transaction may choose to downgrade locks it holds, consequently inheriting them in retained modes to the subtransaction. This concept allows that transactions may see uncommitted data changes of its parents. However, this may not lead to inconsistencies since the commit of a subtransaction depends on the commit of all its parents up to the root transaction.

Controlled downward inheritance of locks in the context of multiple parent transactions immediately leads us to the issue of multiple inheritance of locks. The question is whether there are problems similar to name conflicts in multiple inheritance in class hierarchies or type hierarchies [22]. At a first glance, one might think that such problems may not occur with retained locks inherited from several parents since a certain lock can be held at most by a single transaction. However, since retained locks may also result from upward inheritance of, e.g., sibling subtransactions, there are situations where different types of retained locks on the same object might be inherited from different parents (for an example see below). In these cases, the most restrictive lock is selected for the subtransaction. Again, due to the abort-dependency of the subtransaction to both parent transactions this may not lead to inconsistencies.

Locking Rules. In order to realize these semantics, only the first two (generalized) locking rules of [12], which are given in the following, have to be adapted accordingly. Note, that the term ancestor denotes all direct and indirect parent transactions.

Rule 1: Transaction T may acquire a lock in mode M or upgrade a lock it holds to mode M if no other transaction holds the lock in a mode conflicting with M, and all transactions that retain the lock in a mode conflicting with M are ancestors or *siblings* of T.

Rule 2: When subtransaction T commits, *all parents* of T inherit *those locks (held and retained) of T which have been formally held by T or one of its subtransactions*. In addition, each parent retains the locks in the same mode as T held or retained them before *unless the parent has already a retained lock on the same object in a more restrictive mode*.

Rule 3: When a top-level transaction commits, it releases all locks held or retained.

Rule 4: When a (top-level or sub)transaction aborts, it releases all locks it holds or retains. If any of its superiors hold or retain any of these locks, they continue to do so.

Rule 5: Transaction T, holding a lock in mode M, may downgrade the lock to a less restrictive mode M′. After downgrading the lock, T retains it in the original mode M.

3.3 Example

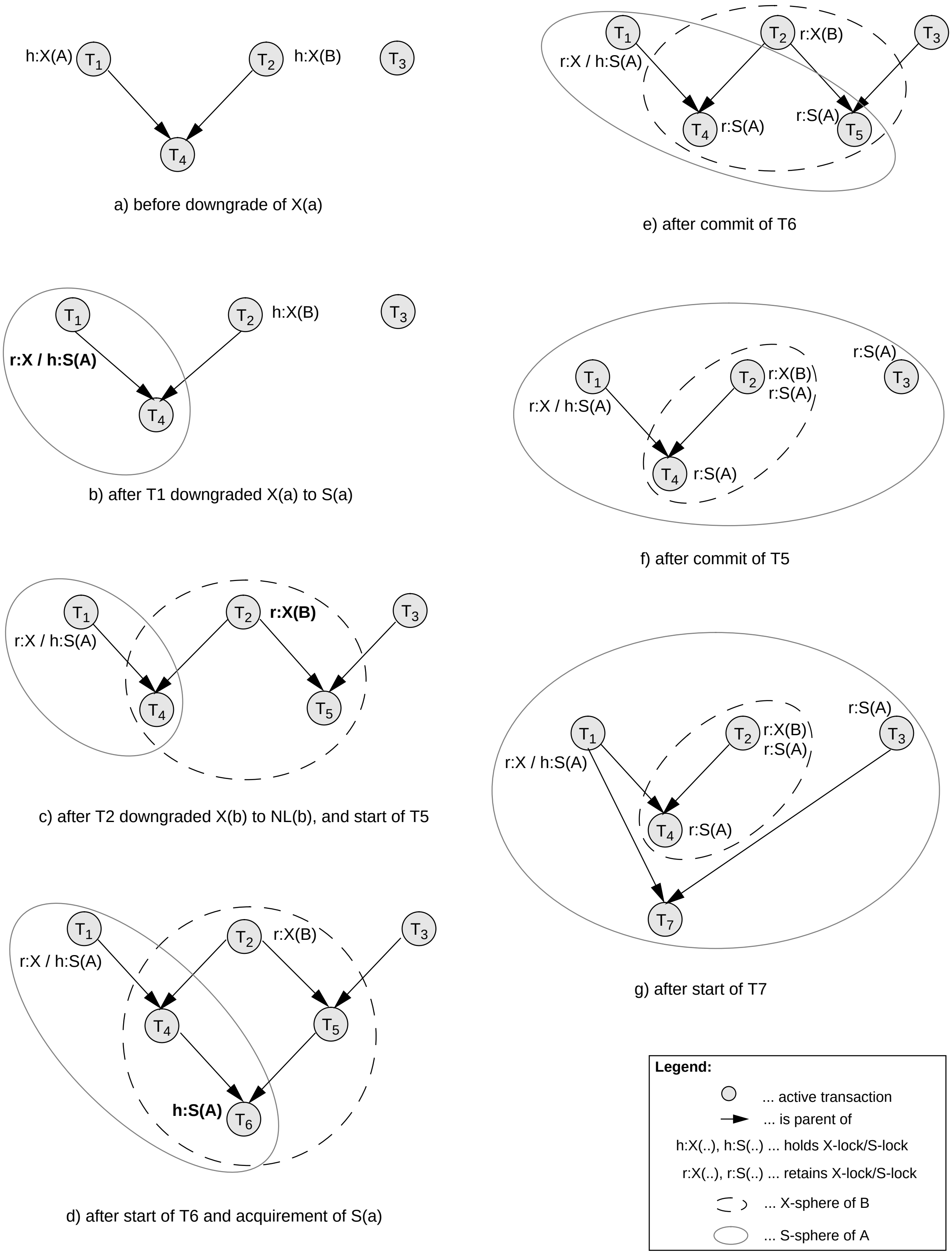
To make things clearer, upward inheritance and controlled downward inheritance of locks are illustrated in the following abstract example. The scenario starts with four transactions T1, T2, T3, and T4 which is a subtransaction of T1 and T2. T1 holds a X-lock on object A, and T2 holds a X-lock on object B (Fig. 4.a). T1 first downgrades the X lock on A to an S lock, thus inheriting it to T4. Consequently, it is now retaining the lock in X mode and holding it in S mode (Fig. 4.b). Since T4 is now in the S-sphere of the object A, it may acquire an S-lock on A which is compatible to the S-lock in T1. T2 and T3, on the contrary, may not acquire any lock on A. In the next step T2 downgrades the X-lock on B to a NL-lock, i.e., no lock. Additionally, T5, a new subtransaction of T2 and T3, is started. Consequently, the X-lock is retained in T2 and inherited to T4 and T5 (Fig. 4.c). T4 and T5 now together start a new common subtransaction T6. Both retained X-locks are automatically inherited to T6. T6, being in the S-sphere of A, acquires a S-lock on that object (Fig. 4.d). Fig. 4.e shows the situation after commit of T6: The S-lock on A is inherited upwards to both parent transactions T4, and T5, which then automatically retain the lock. After commit of T5, the retained S-lock is further inherited to T2 and T3 (Fig. 4.f). Note that, since T5 didn't change or read object B and thus never held a lock on B, the retained X-lock on B need not be inherited upwards to T3. The only changes on object B may have been done in T2 and thus may be externalized upon commit of T2 even if T3 later aborts, undoing the changes of T5.

Assuming finally, that T1 and T3 now start a common subtransaction T7, the problem of multiple inheritance of retained locks arises. As stated above, in this case the most restrictive lock is chosen. Thus, the retained X-lock on A, r:X(A) is inherited to T7 (Fig. 4.g).

4 Applying Multi-Parent Subtransactions

After having described the multi-parent subtransaction model in detail, we will now show how this approach is applied to support the specification of subtransaction modes for several components of a composite event which are signaled in different transactions. In order to illustrate the specification of a subtransaction mode as well as the resulting implications for the execution of event signaling transactions and rule transactions, the example introduced in Subsection 2.2 is revisited. It has to be noted, that, in TriGS, transaction modes specifying the transactional relationships between component event transactions and rule transactions are not restricted to (dependent) subtransactions but include also other transaction modes such as independent top-level transactions. For a discussion of different combinations of these transaction modes we refer to [17].

Transaction Mode Clause. To specify transaction modes for every relationship between a single parent transaction, i.e., an event signaling transaction, and its child transaction, i.e., the rule transaction, separately, the syntax of the rule specification language of TriGS has been extended by a `TRANSACTION MODES` clause. To refer to distinct component events of the event selector, the same referencing-by-name mechanism as introduced in Section 2.2 is used. With this transaction mode clause both, transaction parallelism and transaction dependencies can be specified for a certain rule with respect to each of its component events. Transaction parallelism defines whether the rule transaction has to be executed serial (keyword `SERIAL`) or parallel (keyword `PARALLEL`). Concerning transaction dependencies, the keyword SUBT defines that the rule transaction has to be executed within a subtransaction (serial or parallel). Other transaction modes, which are not concerned in this paper, like the execution of a rule within an independent top-level transaction, can be specified by means of additional keywords (cf. [17]). Applying transaction modes to our example rule of Section 2.2 leads to the following extended rule specification in TriGS:



a) before downgrade of X(a)

b) after T1 downgraded X(a) to S(a)

c) after T2 downgraded X(b) to NL(b), and start of T5

d) after start of T6 and acquirement of S(a)

e) after commit of T6

f) after commit of T5

g) after start of T7

Figure 4: Upward and Downward Inheritance of Locks

```
DEFINE RULE reserveRentalCar AS
ON e1:(PRE(Hotel, reserve:roomRequest for:interval
               customer:aCust) AND
   e2:PRE(Airline, book:destination for:interval
               customer:aCust)) DO
IF checkCustomer /* regular customer &
               accepts cheap rental car offer */ THEN
EXECUTE rentalOffice reserve:compact for:e1.interval
               customer:aCust
TRANSACTION MODES((e1:PARALLEL SUBT, e2:PARALLEL SUBT))
END RULE reserveRentalCar.
```

Two transaction modes define the semantics of the rule transaction, i.e., the car rental transaction, wherein condition and action are executed with respect to the event signaling parent transactions, which are the room reservation transaction and the flight reservation transaction. For both component events a parallel subtransaction is specified within the `TRANSACTION MODES` clause. The abort dependency property of subtransactions realizes that the car rental has only to be done if both, booking and room reservation, are successful. If one of the parent transactions aborts, the car rental transaction has to abort, too. The commit-dependency property in this example avoids that updates to the customer's set of bookings (cf. Fig 5.a) within one or both event signaling transactions might get lost due to an update of the same set within the rule transaction.

As will be seen in the following run-time scenario, the two subtransaction modes are able to express exactly the transaction semantics for this example as requested in Subsection 2.2.

Run-Time Scenario. Figure 5.a shows those parts of the object model of the logically centralized database schema for our example, on which the locks necessary for a proper execution of our reservation transactions are set. The notation is based on [21]. Figure 5.b shows the sequence of method calls within each transaction, the spawning of a rule transaction, and acquirement and downgrading of locks. Since two-phase-locking is assumed locks are automatically released at the end of a transaction. Note, that time progresses from left to right. The letters a to e reference the different steps of upward and downward inheritance including the resulting spheres as illustrated in Figure 6.

In the following the execution of the three transactions is described in a chronological order.

(1) First, the flight reservation transaction signals a book event. According to the specification of a parallel transaction mode, this transaction may continue its execution immediately after signaling this first component event. Within the book method, an S-lock (S(aCust)) on the customer is acquired in order to read general information of the customer like name and address. The object `aCust` is an instance of class `Customer` shown in the object model in Figure 5.a.

(2) Upon finishing the method `book`, the transaction executes the method `addBooking` which adds a new booking instance to the customer's set of bookings (cf. multi-valued relationship `bookings` between the class `Customer` and the class `Reservation` in Figure 5.a) and therefore acquires an X-lock (X(bookings)) on this set. The actual locking situation is depicted in Figure 6.a.

(3) Parallel to the flight reservation transaction, the room reservation transaction signals a `reserve` event which is the triggering event for the car reservation transaction. Due to the parallel transaction mode it proceeds immediately. Following the specification of the transaction modes, the car reservation transaction itself is executed as a subtransaction with respect to both parent transactions.

(4) Figure 6.b shows the locking situation after both, the `reserve` method of the room reservation transaction and the method `checkCustomer` of the car rental transaction, have acquired an S-lock on the customer.

(5) When the room reservation transaction finishes the `reserve` method, it tries to acquire a X-lock on the customer's set of bookings. However, since the flight reservation transaction is already holding that lock, the room reservation transaction has to wait for it depicted by the dotted transaction line in Figure 5.b.

(6) As soon as the flight reservation transaction has finished `addBooking`, it downgrades the X-lock on bookings (cf. X→NL in Figure 5.b) which is no longer needed for the last method of this transaction, the method `notify`, and inherits it to the car reservation transaction. The resulting sphere of the retained lock is shown in Figure 6.c. The car reservation transaction, thus, is able to acquire the lock upon requesting it (Figure 6.d).

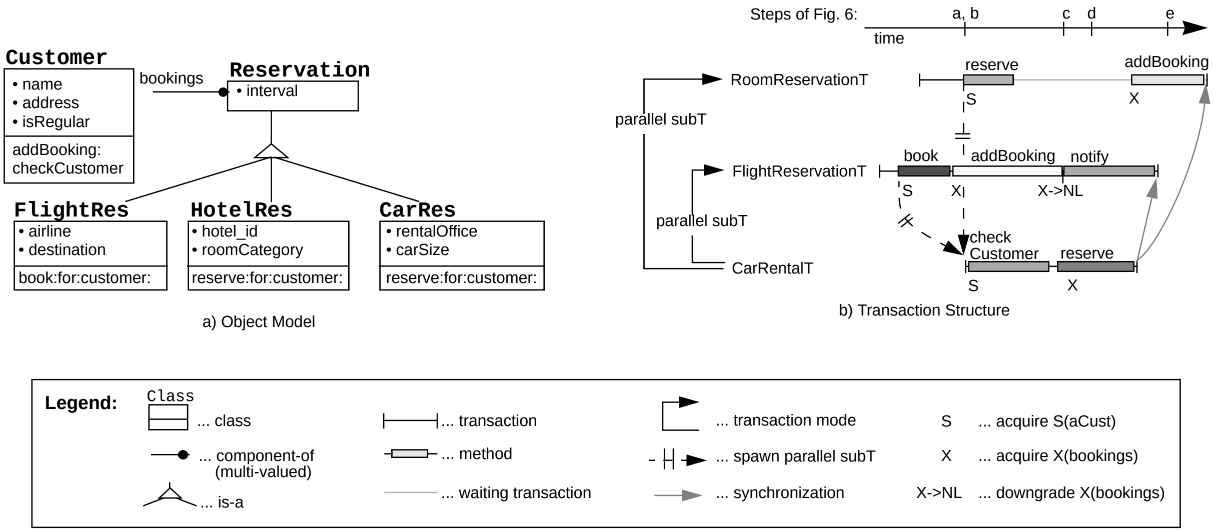


Figure 5: The Booking Example

(7) Upon commit of the rule transaction, the X-lock is inherited upwards in a retained mode to both, the flight reservation transaction and the room reservation transaction. Thus, both transactions have the potential to acquire the X-lock on bookings (Fig. 6.e). The S-lock is not inherited upwards since already held by both parents.

(8) Being the first requesting that lock, the room reservation transaction gets the X-lock on bookings and finally executes the method `addBooking` analogously to the flight reservation transaction. (This is not shown in Figure 6).

5 Conclusion

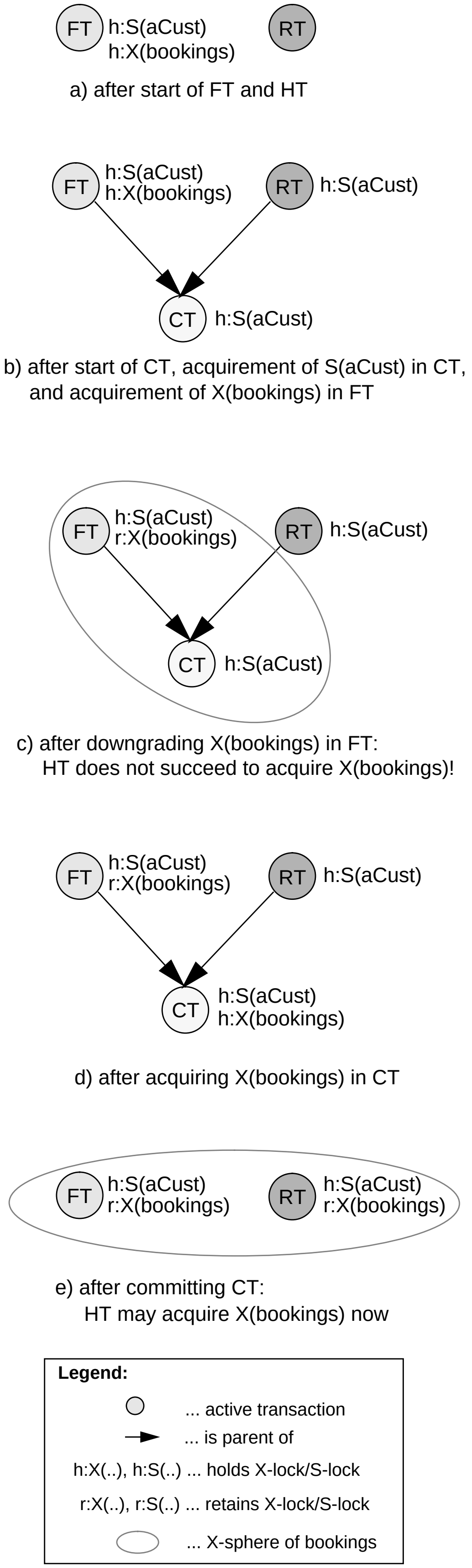
This paper has shown that in the presence of composite events there is a need to apply transaction modes not only to the triggering transaction but also to some or all of the event signaling transactions. Our approach allows to explicitly specify transaction modes for each component event in the context of a single rule. The main purpose of this paper is to show the implications of this approach on the nested transaction model, which has been suggested as appropriate transaction model for AOODBSs. The original nested transaction model cannot cope with situations where a subtransaction mode is used for more than one component event. Thus, it has been extended in order to support multi-parent subtransactions. Special emphasis was given on the development of a proper locking protocol. The proposed extensions allow to combine the reliability and flexibility of the nested transaction model with the expressive power of composite events.

5.1 Related Work

Only some systems consider transaction modes in the context of composite events. Most important among these are REACH [3], SAMOS [9], Sentinel [4], and Ode [10, 18].

Figure 7 compares our approach to these systems with respect to their transaction support according to the following criteria which are factored out from the discussion in Subsection 2.2:

Origin of the component events. This criterion deals with the question whether composite events may occur across transaction boundaries (cf. Figure 1.b-d). In the simplest case, which is supported by all systems, they might occur within a single transaction. On the other hand, they may occur within multiple transactions that are executed within a single thread of control. Finally, they might be raised from within transactions executed within different threads of control.



a) after start of FT and HT

b) after start of CT, acquirement of S(aCust) in CT, and acquirement of X(bookings) in FT

c) after downgrading X(bookings) in FT:
HT does not succeed to acquire X(bookings)!

d) after acquiring X(bookings) in CT

e) after committing CT:
HT may acquire X(bookings) now

Figure 6: Lock Inheritance in the Booking Example

	Ode	REACH	Sentinel	SAMOS	TriGS
Origin of the Component Events					
- single transaction	✓	✓	✓	✓	✓
- multiple transactions within a single thread	✓	✓	✘	✓	✓
- multiple transactions within different threads	✓	✓	✘	✓	✓
Nested Transaction Model					
- single-parent subtransaction	✘	✘	✓	✓	✓
- multi-parent subtransaction	✘	✘	✘	✘	✓
Reference Point for a Transaction Mode					
- triggering transaction only	✓	✘	✓	✓	✓
- all event signaling transactions	✘	✓	✘	✘	✓
- a specific event signaling transaction	✘	✘	✘	✘	✓
Event-Specific Transaction Modes	✘	✘	✘	✘	✓

Figure 7: Transaction Support for Composite Events

Nested transaction model. This criterion shows whether the nested transaction model is used by the various systems in order to execute a rule within a subtransaction of the event signaling transaction(s). Two variations of the nested transaction model, single-parent subtransactions and multi-parent subtransactions, are examined.

Reference point for a transaction mode. In the context of component events arising from different transactions, it has to be specified whether a specific transaction modes refers to the triggering transaction only, to all event signaling transactions, or to a specific event signaling transaction.

Event-specific transaction modes. This criterion refers to the issue whether only a single transaction mode can be specified which is applied to all event signaling transactions, or whether for each component event a separate transaction mode may be specified. This is tightly coupled to the question of the reference point. Only if a single transaction mode can be related to a distinct event-signaling transaction, different event-specific transaction modes can be supported.

It can be subsumed that, besides TriGS, none of these approaches supports transaction modes related to specific events, and none of these have a multi-parent subtransaction model. Concerning the nested transaction model, it is interesting to note that Sentinel‘s locking protocol allows for five different holdmodes of locks including a retained mode in order to avoid searching the whole transaction tree for conflicting locks when requesting a lock. REACH and Ode do not use a nested transaction model, whereas SAMOS builds on the nested transaction model of the underlying database system which does not allow parent/child parallelism nor parallelism between siblings.

5.2 Open Issues

Concerning our approach of event-specific transaction modes, and in particular multi-parent subtransactions, there are a lot of issues which have not yet been fully investigated. In the following, a few of them are briefly outlined.

Looking at the example in Section 2.2, the attentive reader might recognize that the commit-dependency between event signaling transactions and the rule transaction might lead to a problem concerning the termination of the event signaling transactions. Suppose that the room reservation transaction is not started before the flight reservation transaction is ready to commit, or vice versa. This means that the first component event has already been signaled whereas the second, i.e., the triggering event is still pending. The question arises whether the flight reservation should wait for the room reservation — which need not occur at all — and if yes, how long it should wait. If the room reservation transaction is never executed, the flight reservation transaction would wait for an indefinite period of time. This problem not only occurs in the context of a subtransaction which is specified to be parallel to each event signaling transaction but is even worse in the context of a subtransaction which is defined to be serial to some event signaling transaction. In the serial case all these event signaling transactions would have to wait for the occurrence of the triggering event not knowing when it will occur and whether it will occur at all. One solution to this problem, which was suggested by

REACH [3], is to disallow the specification of a serial transaction mode in the context of composite events or to allow a serial transaction mode for the triggering event only. Our approach adheres to another solution, originally suggested by Sentinel [4], namely the use of a time-out mechanism which can be realized by using a relative or absolute event specification. Concerning our example, let us assume that the policy is that both reservation transactions have to be started within one hour. Otherwise, the special offer of a cheap car reservation cannot be claimed anymore. For this, the composite event of our car reservation rule simply has to be extended with two relative time events:

```
...
ON  e1 AND e2 AND NOT REL(e1,1HR) AND NOT REL(e2,1HR) DO
...
```

Another problem in the context of multi-parent transactions not discussed so far are modifications within a parent transaction based on committed values of a child transaction, which later might be aborted due to the abort-dependency to another parent transaction. Upon abort, all such modifications would have to be rolled back within the parent. In case that such a parent transaction also had already committed, cascading aborts have to be performed. In order to avoid keeping track of which modifications are based on which child transaction, vital subtransactions [5] would have to be used within the whole transaction graph. A vital subtransaction specifies the parent to be abort-dependent on the child. This means that an abort of a single transaction T1 would result in aborts of all transactions which are direct or indirect subtransactions of T1 or, due to the usual subtransaction semantics, direct or indirect parents of these subtransactions. The only transactions that would not be affected by these aborts are the direct and indirect parents of T1. A reasonable approach to this problem still has to be found.

Further issues that have to be solved include optimization, design guidelines for specifying adequate transaction modes, the influence of different consumption modes on our approach, and the incorporation of different kinds of primitive events.

References

[1] P.C. Attie, M.P. Singh, A. Sheth and M. Rusinkiewicz, “Specifying and Enforcing Intertask Dependencies,” in *Proc. of the 19th VLDB,* Dublin, Ireland, 1993.

[2] C. Beeri and T. Milo, “A Model for Active Object Oriented Database,” in *Proc. of the 17th Int. Conference on Very Large Databases (VLDB),* Barcelona, 1991.

[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 Int. Conf. on Data Engineering,* Taipeh, 1995.

[4] S. Chakravarthy, V. Krishnaprasad, E. Anwar and S.-K. Kim, “Composite Events for Active Databases: Semantics, Contexts and Detection,” in *Proc. of the 20th Int. Conference on Very Large Data Bases (VLDB’94), ,* Santiago, Chile, 1994.

[5] P.K. Chrysanthis and K.Ramamritham, “ACTA: The SAGA Continues,” in *Database Transaction Models for Advanced Applications,* ed. A. Elmagarmid, Morgan Kaufmann, 1992.

[6] C. Collet, T. Coupaye and T. Svenson, “NAOS - Efficient and modular reactive capabilities in an Object-Oriented Database System,” in *Proc. of the 20th Int. Conference on Very Large Data Bases (VLDB’94),* Santiago, Chile, 1994.

[7] U. Dayal, M. Hsu and R. Ladin, “Organizing Long-Running Activities with Triggers and Transactions,” in *Proc. of the 1990 ACM SIGMOD Int. Conference on Management of Data,* Atlantic City, NJ, 1990 .

[8] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner and K. Salem, “Modeling Long-Running Activities as Nested Sagas,” IEEE Bulletin of the Technical Committee on Data Engineering, vol. 14, no. 1, 1991.

[9] S. Gatziu, *Events in an Active Object-Oriented Database System*, Kovac Verlag, Hamburg, 1995.

[10] N.H. Gehani and H.V. Jagadish, “Active Database Facilities in ODE,” in *Active Database Systems - Triggers and Rules for Advanced Database Processing*, ed. J. Widom and S. Ceri , Morgan Kaufmann Publishers Inc., San Francisco, California, 1996.

[11] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.

[12] T. Härder and K. Rothermel, “Concurrency Control Issues in Nested Transactions,” VLDB Journal, vol. 2, no. 1, pp. 39-74, 1993.

[13] M. Hsu, R. Ladin and D.R. McCarthy, “An Execution Model For Active Database Management Systems,” in *Proc. of the 3rd Int. Conf. On Data and Knowledge Bases: Improving usability and responsiveness*, ed. U. Dayal , Morgan Kaufmann Publishers, Inc., Jerusalem, Israel, June 1988.

[14] G. Kappel, S. Rausch-Schott, W. Retschitzegger and S. Vieweg, “TriGS: Making a Passive Object-Oriented Database System Active,” in *Journal of Object-Oriented Programming (JOOP)*, vol. 7, pp. 40-51, July/August 1994.

[15] G. Kappel, S. Rausch-Schott and W. Retschitzegger, “Beyond Coupling Modes - Implementing Active Concepts on Top of a Commercial OODBMS,” in *International Symposium on Object-Oriented Methodologies and Systems (ISOOMS)*, ed. E.Bertino and S.Urban, Springer LNCS 858, 1994.

[16] G. Kappel, P. Lang, S. Rausch-Schott and W. Retschitzegger, “Workflow Management Based on Objects, Rules, and Roles,” IEEE Bulletin of the Technical Committee on Data Engineering, vol. 18, no. 1, March 1995.

[17] G. Kappel, S. Rausch-Schott, W. Retschitzegger and M. Sakkinen, “Multi-Parent Subtransactions Covering the Transactional Needs of Composite Events,” in *International Workshop on Advanced Transaction Models and Architectures (ATMA’96)*, Goa (India), September 1996.

[18] D.F. Lieuwen, N. Gehani and R. Arlein, “The Ode Active Database: Trigger Semantics and Implementation,” in *Proc. of the 12th Int. Conf. on Data Engineering*, pp. 412-420, New Orleans, Louisiana, 1996.

[19] E. Moss, *Nested Transactions*, The MIT Press, Cambridge, MA, 1985.

[20] M.T. Özsu, “Transaction Models and Transaction Management in Object-Oriented Database Management Systems,” in *Advances in Object-Oriented Database Systems* , ed. A. Dogac, M.T. Özsu, A. Biliris, and T. Sellis, Springer Verlag NATO ASI Series F130, 1994.

[21] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modelling and Design*, Prentice-Hall, 1991.

[22] M. Sakkinen, “Disciplined Inheritance,” in *ECOOP 89*, ed. S. Cook, pp. 39-56, Cambridge University Press, July 1989.

[23] E. Simon and A. Kotz-Dittrich, “Promises and Realities of Active Database Systems,” in *Proc. of the 21st Conf. on Very Large Data Bases (VLDB)*, Zurich, Switzerland, 1995.

From gerti@ifs.uni-linz.ac.at Wed Nov 6 17:50:39 1996 Return-Path: ¡gerti@ifs.uni-linz.ac.at¿ Received: from alijku04.edvz.uni-linz.ac.at by meta.niimm.spb.su (SMI-8.6/SMI-SVR4) id RAA28040; Wed, 6 Nov 1996 17:50:04 +0300 Received: from ifs.uni-linz.ac.at (lion.ifs.uni-linz.ac.at) by alijku04.edvz.uni-linz.ac.at with SMTP id AA24828 (5.67c/IDA-1.5 for ¡boris@meta.niimm.spb.su¿); Wed, 6 Nov 1996 15:48:05 +0100 Received: by ifs.uni-linz.ac.at (4.1/SMI-4.1) id AA19804; Wed, 6 Nov 96 15:48:12 +0100 Date: Wed, 6 Nov 96 15:48:12 +0100 From: gerti@ifs.uni-linz.ac.at (Gerti Kappel) Message-Id: ¡9611061448.AA19804@ifs.uni-linz.ac.at¿ To: boris@meta.niimm.spb.su Subject: 3.mail - bib.tex X-Mozilla-Status: 0001 Content-Length: 5283

References

[1] P.C. Attie, M.P. Singh, A. Sheth and M. Rusinkiewicz, “Specifying and Enforcing Intertask Dependencies,” in *Proc. of the 19th VLDB*, Dublin, Ireland, 1993.

[2] C. Beeri and T. Milo, “A Model for Active Object Oriented Database,” in *Proc. of the 17th Int. Conference on Very Large Databases (VLDB)*, Barcelona, 1991.

[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 Int. Conf. on Data Engineering*, Taipeh, 1995.

[4] S. Chakravarthy, V. Krishnaprasad, E. Anwar and S.-K. Kim, “Composite Events for Active Databases: Semantics, Contexts and Detection,” in *Proc. of the 20th Int. Conference on Very Large Data Bases (VLDB’94)*, , Santiago, Chile, 1994.

[5] P.K. Chrysanthis and K.Ramamritham, “ACTA: The SAGA Continues,” in *Database Transaction Models for Advanced Applications*, ed. A. Elmagarmid, Morgan Kaufmann, 1992.

[6] C. Collet, T. Coupaye and T. Svenson, “NAOS - Efficient and modular reactive capabilities in an Object-Oriented Database System,” in *Proc. of the 20th Int. Conference on Very Large Data Bases (VLDB’94)*, Santiago, Chile, 1994.

[7] U. Dayal, M. Hsu and R. Ladin, “Organizing Long-Running Activities with Triggers and Transactions,” in *Proc. of the 1990 ACM SIGMOD Int. Conference on Management of Data*, Atlantic City, NJ, 1990 .

[8] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner and K. Salem, “Modeling Long-Running Activities as Nested Sagas,” IEEE Bulletin of the Technical Committee on Data Engineering, vol. 14, no. 1, 1991.

[9] S. Gatziu, *Events in an Active Object-Oriented Database System*, Kovac Verlag, Hamburg, 1995.

[10] N.H. Gehani and H.V. Jagadish, “Active Database Facilities in ODE,” in *Active Database Systems - Triggers and Rules for Advanced Database Processing*, ed. J. Widom and S. Ceri , Morgan Kaufmann Publishers Inc., San Francisco, California, 1996.

[11] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.

[12] T. Härder and K. Rothermel, “Concurrency Control Issues in Nested Transactions,” VLDB Journal, vol. 2, no. 1, pp. 39-74, 1993.

[13] M. Hsu, R. Ladin and D.R. McCarthy, “An Execution Model For Active Database Management Systems,” in *Proc. of the 3rd Int. Conf. On Data and Knowledge Bases: Improving usability and responsiveness*, ed. U. Dayal , Morgan Kaufmann Publishers, Inc., Jerusalem, Israel, June 1988.

[14] G. Kappel, S. Rausch-Schott, W. Retschitzegger and S. Vieweg, “TriGS: Making a Passive Object-Oriented Database System Active,” in *Journal of Object-Oriented Programming (JOOP)*, vol. 7, pp. 40-51, July/August 1994.

[15] G. Kappel, S. Rausch-Schott and W. Retschitzegger, “Beyond Coupling Modes - Implementing Active Concepts on Top of a Commercial OODBMS,” in *International Symposium on Object-Oriented Methodologies and Systems (ISOOMS)*, ed. E.Bertino and S.Urban, Springer LNCS 858, 1994.

[16] G. Kappel, P. Lang, S. Rausch-Schott and W. Retschitzegger, “Workflow Management Based on Objects, Rules, and Roles,” IEEE Bulletin of the Technical Committee on Data Engineering, vol. 18, no. 1, March 1995.

[17] G. Kappel, S. Rausch-Schott, W. Retschitzegger and M. Sakkinen, “Multi-Parent Subtransactions Covering the Transactional Needs of Composite Events,” in *International Workshop on Advanced Transaction Models and Architectures (ATMA’96)*, Goa (India), September 1996.

[18] D.F. Lieuwen, N. Gehani and R. Arlein, “The Ode Active Database: Trigger Semantics and Implementation,” in *Proc. of the 12th Int. Conf. on Data Engineering*, pp. 412-420, New Orleans, Louisiana, 1996.

[19] E. Moss, *Nested Transactions*, The MIT Press, Cambridge, MA, 1985.

[20] M.T. Özsu, "Transaction Models and Transaction Management in Object-Oriented Database Management Systems," in *Advances in Object-Oriented Database Systems* , ed. A. Dogac, M.T. Özsu, A. Biliris, and T. Sellis, Springer Verlag NATO ASI Series F130, 1994.

[21] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modelling and Design*, Prentice-Hall, 1991.

[22] M. Sakkinen, "Disciplined Inheritance," in *ECOOP 89*, ed. S. Cook, pp. 39-56, Cambridge University Press, July 1989.

[23] E. Simon and A. Kotz-Dittrich, "Promises and Realities of Active Database Systems," in *Proc. of the 21st Conf. on Very Large Data Bases (VLDB)*, Zurich, Switzerland, 1995.