NESTED TRANSACTIONS: REVIEWING THE COHERENCE CONTRACT *

Rachid Guerraoui
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
CH-1015, Lausanne, Switzerland
e-mail : guerraoui@di.epfl.ch

Abstract: In this paper we review nested transactions from the coherence point of view. We present the contract that has to be established between a distributed database system and its nested transaction programmers, in order to achieve database coherence. Terms of the contract are described through the underlying properties of nested transactions, which we define as *nested-all-or-nothing, nested-consistency, nested-isolation* and *nested-durability*. By providing a precise semantic of nesting, these properties enhance the safe use of nested transactions in distributed database systems.

1. INTRODUCTION

Transactions have been widely accepted as a fundamental principle to preserve database coherence when coping with concurrency and failures [1,3,9,12]. In their primeval model, transactions were viewed as monolithic sequences of object requests, which satisfy the "ACID" properties: *(A) all-or-nothing, (C) consistency, (I) isolation* and *(D) durability* [8]. According to this monolithic model, database coherence is preserved as long as some system underlying mechanism guarantees transaction atomicity, i.e. the A, I and D properties for all transactions, while programmers ensure the C property for their transactions (without taking into account neither concurrency nor potential failures). From a programmer point of view, this coherence contract reduces the difficult problem of maintaining the coherence of a concurrent system subject to failures, into the easier problem of maintaining the coherence of sequential failure-free subsystems, namely transactions.

Nevertheless, because of the rigidity nature of atomicity, the original transaction model is not suitable for a general use in distributed database systems [14]. Long running distributed transactions may monopolize shared objects during non-acceptable periods of time, and local failures cannot be masked. Nesting transactions has proved particularly useful in distributed database systems as it provides a simple mechanism for obtaining concurrency within a transaction and for limiting the effects of a failure to a small part of a transaction [11,5,6,10].

Although many synchronization and failure recovery algorithms have been proposed in the literature for nested transactions [11,2,3,9], little has been done on their consistency requirements. In fact, since nested transactions are not atomic in the original sense anymore, ensuring the consistency of each individual failure-free transaction is no longer a sufficient condition to preserve coherence. Hence, the safe use of nested transactions goes through reviewing their coherence contract.

This paper reviews this contract by defining the "N-ACID" properties of nested transactions: *(N-A) nested-all-or-nothing, (N-C) nested-consistency, (N-I) nested-isolation* and *(N-D) nested durability.* When the N-A, N-I and N-D are guaranteed by some database system underlying mechanism, programmers must ensure the N-C property of transaction trees to preserve database coherence. When defining these properties, we have considered a general nesting model which encompasses various nested transaction models [1, 2, 4, 10]. Consequently, we believe that our coherence contract may be applied to all those models.

*Appears in Journal of Information Systems, 1995. A preliminary version appeared in the Proceedings of the International Symposium on Computer and Information Systems, 1993.

The paper is organized as follows. Next section reminds the coherence contract in the original transaction model and discusses the coherence issues raised by nesting transactions. Section 3 describes our general nested transaction model. Section 4 presents the *nested-atomicity*, and *nested-consistency* properties, and shows how, when combined, these properties preserve database coherence. Section 5 summarizes the main ideas of the paper and concludes with some general remarks.

2. COHERENCE AS A CONTRACT

2.1. The "ACID" properties

The basic idea in transaction processing is to group database object requests which preserve coherence, inside larger units called transactions of which atomicity is ensured by some system underlying mechanism. Broadly speaking, atomicity means that even in the face of concurrency and failures, everything happens as if each transaction was executing alone in a failure-free way. More accurately, transactions have been defined as sequences of object requests which satisfy the "ACID" properties [8]:

- *(A) All-or-nothing* (or failure atomicity): either a transaction is committed and has its desired effects, or it is aborted and has no effect.
- *(C) Consistency*: when executed (i) alone, i.e. without concurrency, (ii) in a failure-free environment, and (iii) from coherent objects' states, a transaction produces coherent replies and coherent final states.
- *(I) Isolation* (or concurrency atomicity): the intermediate states of the objects manipulated by a transaction are not visible outside that transaction.
- *(D) Durability* (or permanence of effects): the effects of a committed transaction are not undone by a failure.

Database coherence is preserved as long as the system guarantees transaction atomicity, i.e. the properties *A, I* and *D* , while each programmer ensures the property *C* of his transactions. This agreement between the programmers and the database system is what we will call the coherence contract in this paper (figure 1).

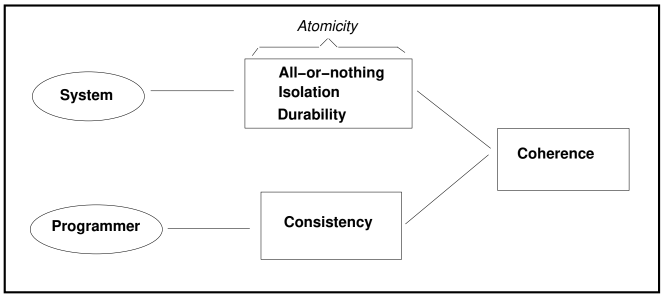

Figure 1: The original coherence contract

2.2. Nesting transactions

To better support parallelism and limit the effects of failures, it is convenient to break the monolithic view of transactions by allowing their nesting [11,5,6,10]. Beside requests, a nested transaction can contain any number of subtransactions. Each subtransaction may at its turn contain any number of subtransactions and so on. To describe nesting relations, family relationship terminology is used, i.e. "parent", "child", "ancestor", "descendent", "sibling", etc.

Figure (2) shows a set of nested transactions $\{T1, T11, T12, T2, T21, T22\}$. This set is made of two "transaction trees", i.e. $\{T1, T11, T12\}$, and $\{T2, T21, T22\}$. A transaction tree is a set compound of a top-level transaction (without parent), e.g. $T1$ or $T2$, together with its descendents. Consider for instance the transaction tree $\{T1, T11, T12\}$. $T1$ contains two requests $req11$ and $req12$ as well as two child transactions

$T11$ and $T12$. $T11$ contains two requests $req111$ and $req112$, while $T12$ contains a request $req121$. Two transactions are said to be independent if none of them is the ancestor of the other, e.g. $T11$ and $T12$ are independent, whereas $T1$ and $T11$ are not.

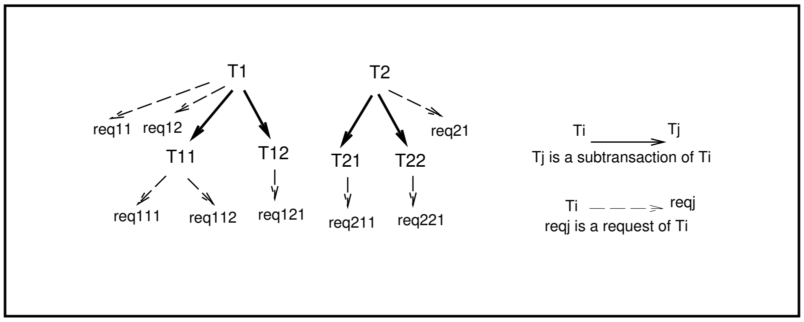

Figure 2: Nested structure

Among child transactions of the same parent, some may execute sequentially while others may execute concurrently. The latters perform with the guarantee that they will appear to have run in isolation with respect to their parent and to their sibling. Hence, intra-transaction concurrency can safely be introduced through child transactions. In addition, the abort of a child transaction does not entail the abort of its parent. This allows to mask local failures by confining them into subtransactions. Atomic replicated database updates [11], atomic RPC [10], and atomic asynchronous RPC [5] are examples of beneficial uses of nested transactions.

However, the flexibility of nested transactions does not come for free; nested transactions are no longer atomic in the original sense:

1. A child transaction may read and modify the intermediate states of objects accessed by its parent. Hence, a parent transaction does not satisfy the *isolation* property.
2. A child transaction may fail without forcing its parent to do so. Therefore, a parent transaction does not satisfy the *all-or-nothing* property.
3. Even though a child transaction terminates successfully, its effects are not permanent until its top-level ancestor has committed. Thus, a child transaction does not satisfy the *durability* property.

Since atomicity is no longer guaranteed, ensuring the consistency of each individual failure-free transaction is not a sufficient condition to preserve database coherence. Given the nested structure of figure 2 for example, $T11$ and $T12$ may read and modify the intermediate states of the objects accessed by $T1$. To preserve coherence, it is not sufficient to ensure that $T1$'s requests, i.e. $req11$ and $req12$, produce coherent results and coherent final states. It is necessary to take into account the interferences with $T11$ and $T12$, as well as their potential aborts. Moreover, when dealing with the consistency of either $T11$ or $T12$, one cannot assume coherent initial objects' states: these states may reflect some intermediate, thus incoherent, effects of $T1$.

To redefine the coherence contract for nested transactions, we first precise our general nesting model in section 3, then we present the *nested-atomicity* and *nested-consistency* properties in section 4.

3. A GENERAL NESTED TRANSACTION MODEL

The main differences among the various models of nested transactions have to do with whether or not a parent can directly access an object, and whether or not it can execute concurrently with its children. The nested transaction models described in [9,2,10] does not permit a parent transaction to directly access the objects. Such accesses are reserved to leaf transactions which do not contain any child. The model presented in [3] authorizes a parent transaction to directly access objects, but does not allow a parent transaction to run concurrently with its children. Note that we do not consider here the so-called multi-level model [16], which corresponds more to nested object operations than to nested transactions .

When dealing with coherence, we have found it important to consider a general model which encompasses those various "submodels". We have thus considered a nested transaction model which removes those limitations, such that:

1. Each transaction, whatever is its nesting level, can directly access the objects.
2. A parent transaction can execute concurrently with its subtransactions.

3.1. Structure

In our nesting model, a transaction contains a set of requests, as well as a set of child transactions. Each transaction performs its requests in a sequential way, i.e. it addresses a request, waits for a reply, receives the reply, addresses another request etc. Therefore, the set of transaction's requests is totally ordered. Among transaction's children, some may execute concurrently, others sequentially. Since we allow a transaction to perform concurrently with its children, the set composed of requests and child transactions is partially ordered. If a parent transaction aborts, all its children do so. However, the abort of a child transaction does not necessary implies the abort of the parent. We introduce here the notion of "necessary" child transaction: a child transaction is said to be necessary if its abort infers the abort of its parent. In other words, a parent transaction cannot commit unless its subtransactions have done so.

More precisely, we describe a transaction T as a tuple (Q_T, S_T, N_T, O_T) where:

- Q_T is the set of T's object requests.
- S_T is the set of T's child transactions.
- N_T is the restriction of S_T to the necessary child transactions.
- O_T is a total order relation on Q_T, and a partial order relation on $(Q_T \bigcup S_T)$.

Consider for example the transaction tree $\{T1, T11, T12\}$ in figure 2: $Q_{T1} = \{req11, req12\}$, $Q_{T11} = \{req111, req112\}$, $Q_{T12} = \{req121\}$; $S_{T1} = \{T11, T12\}$, $S_{T11} = \emptyset$, $S_{T12} = \emptyset$. We may for example assume that neither $T11$ nor $T12$ are necessary child transactions, i.e. $N_{T1} = \emptyset$, and that only the requests are ordered, e.g. $O_T1(req11, req12)$ and $O_T11(req111, req112)$.

3.2. Executions

As in [15], we depict an execution of a set of nested transactions as a finite sequence of events. Each event is associated both to a transaction and to an object. Significant events are request invocations, replies and termination notifications. A termination notification is a particular request addressed to an object from a transaction, meaning either the abort or the commit of a transaction. The transaction is said to have aborted or committed at that object.

More precisely, we describe an execution $E_{(\psi,\omega)}$, of a set of nested transactions ψ, on a set of objects ω in some initial states, as a tuple $(Q_{E_{(\psi,\omega)}}, R_{E_{(\psi,\omega)}}, A_{E_{(\psi,\omega)}}, O_{E_{(\psi,\omega)}})$:

- $Q_{E_{(\psi,\omega)}}$ is the set of requests addressed to ω's objects.
- $R_{E_{(\psi,\omega)}}$ is the set of corresponding replies.
- $A_{E_{(\psi,\omega)}}$ is the set of termination notifications.
- $O_{E_{(\psi,\omega)}}$ is a total order relation on $V_{E_{(\psi,\omega)}} = (Q_{E_{(\psi,\omega)}} \bigcup R_{E_{(\psi,\omega)}} \bigcup A_{E_{(\psi,\omega)}})$.

Not all sequences of events make sense as observation of nested transaction executions. A transaction starts by a request and does not perform any request after its termination. Hence, the first event associated to a transaction must be a request while the last one must be a termination notification. As we assume non-interruptibility for object operations, there must be no event associated to the same object, between a request and the corresponding reply. In addition, a transaction cannot perform a commit notification if its parent or one of its necessary subtransactions has aborted, and cannot commit at some objects and abort on others. Finally, a correct sequence of events must respect the sequential predefined order O_T of each transaction T. In the following, we consider only the sequences of events which obey these properties. Such sequences are intended to depict the typical transaction execution in our general nesting model.

Figure 3 shows an execution $E_{(\psi,\omega)}$, where $\psi = \{T1, T11, T12, T2, T21, T22\}$ (figure 2). $T1, T11, T2, T21$ and $T22$ commit, whereas $T12$ aborts. When it terminates successfully, a child transaction, e.g. $T11$, reaches a preliminary commit point (p-commit point). The real commit point coincides with the top-level ancestor one, e.g. the commit point of $T1$. Since we assumed that $T12$ is not a necessary child transaction of $T1$, the latter has committed although the former has aborted.

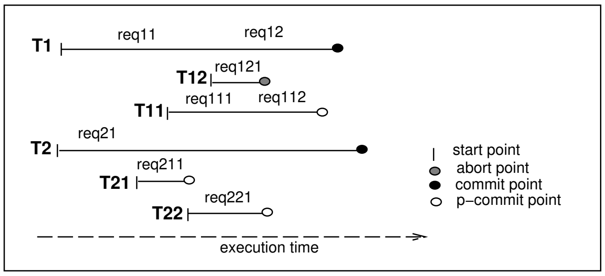

Figure 3: Nested execution

4. *N-ACID* PROPERTIES

4.1. Atomicity

Nested-atomicity, i.e. atomicity of nested transactions, can be described through three properties, which we call *nested-all-or-nothing, nested-isolation and nested-durability*:

- *(N-A) Nested all-or-nothing*: either a transaction is committed and has its desired effects, or it is aborted and has no effect.
- *(N-I) Nested-isolation*: the intermediate states of the objects manipulated by a transaction, are neither visible to its ancestors nor to independent transactions.
- *(N-D) Nested-durability*: the effects of a committed top-level transaction and those of its committed descendents are not undone by a failure.

To gather these properties in a more formal way, we introduce the notions of sequential execution, permanent restriction, and execution equivalence. A sequential execution is one such that independent transactions execute in a sequential way, and each parent transaction waits for its children to complete before pursuiting its execution, i.e. the parent does not perform a request when a child is executing. The permanent restriction of an execution $E_{(\psi,\omega)}$, noted $Committed(E_{(\psi,\omega)})$, is the restriction of $E_{(\psi,\omega)}$ to its committed transactions. Finally, we say that two executions are equivalent iff they return the same replies, i.e. $E1_{(\psi,\omega)} \equiv E2_{(\psi,\omega)} \Leftrightarrow R_{E1_{(\psi,\omega)}} = R_{E2_{(\psi,\omega)}}$.

Figure 4 shows a sequential execution $Es_{(\psi,\omega)}$, where $\psi = \{T1, T11, T12, T2, T21, T22\}$. Each parent transaction, i.e. $T1$ or $T2$, waits for its children to complete before pursuiting its execution. The parent does not perform any request during a child execution. In addition, independent transactions, e.g. $T21$ or $T22$, execute in a sequential way.

A sequential execution may be viewed as a sequence of transaction tree sequential executions, in this figure (4) 4, $Es_{(\psi,\omega)} = Es_{(\psi 1,\omega 1)}; Es_{(\psi 2,\omega 2)}$, where $\psi 1 = \{T1, T11, T12\}$ and $\psi 2 = \{T2, T21, T22\}$. The permanence restriction $Committed(Es_{(\psi,\omega)})$, of $Es_{(\psi,\omega)}$, would be the restriction of $Es_{(\psi,\omega)}$, to its committed transactions, i.e. without $T12$.

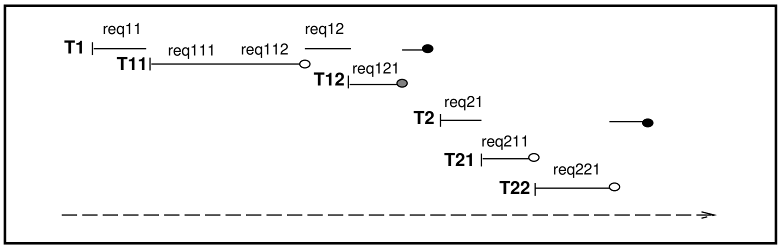

Figure 4: Sequential execution

When satisfied for all transactions, the *nested-isolation* property means that an execution appears as if each transaction was isolated from its ancestors and from independent transactions. In such an execution, independent transactions would appear to execute in a sequential way and each transaction would appear to wait for its child transactions to complete before pursuiting its execution. When satisfied for all transactions, the *nested-all-or-nothing* and the *nested-durability* properties imply that an execution must appear as if it contains only committed transactions. Hence, if all transactions satisfy *nested-atomicity* in an execution $E_{(\psi,\omega)}$, $Committed(E_{(\psi,\omega)})$ will be equivalent to a sequential execution, i.e. to a sequence of transaction tree sequential executions.

4.2. Coherence

If *nested-atomicity* is guaranteed for all transactions, each execution will appear to happen as if each transaction tree executes alone, in a sequential way, from a state produced by a sequence of committed transaction tree sequential executions. The replies and final states produced by each transaction tree will depend on the computation inside the tree. Therefore, it seems natural to consider consistency, as a transaction tree property. We introduce the *nested-consistency* property as follows:

- *(N-C) Nested Consistency:* when it performs a sequential execution from coherent initial states, a transaction tree produces coherent replies and coherent final states.

  Theorem: When combined, the *N-ACID* properties preserve database coherence.

  Proof: The *N-A, N-I* and *N-D* properties imply that for each execution $E_{(\psi,\omega)}$, $R_{Committed(E_{(\psi,\omega)})} = \bigcup_{k=1} R_{Es^k_{(\psi^k,\omega^k)}}$, where each $Es^k_{(\psi^k,\omega^k)}$ denotes a sequential execution of the committed transaction tree ψ^k, on the corresponding object set ω^k.

  Hence, provided that the *N-A, N-I* and *N-D* properties are guaranteed for all transactions, ensuring the coherence of $R_{Committed(E_{(\psi,\omega)})}$ amounts to ensuring the coherence of each $R_{Es^k_{(\psi^k,\omega^k)}}$. This comes down to guarantee that when a nested transaction tree ψ_k performs a sequential execution $Es^k_{(\psi^k,\omega^k)}$ on a set of objects ω^k, it produces, coherent results $R_{Es^k_{(\psi^k,\omega^k)}}$, and coherent final states ω^{k+1}, namely the *N-C* property of the transaction tree ψ_k.

  Therefore, if the *N-A, N-I* and *N-D* properties are guaranteed for all transactions, preserving coherence comes down to ensure the *N-C* property for transaction trees. Consequently, when combined, the *N-ACID* properties preserve coherence.

4.3. The coherence contract revisited

The new coherence contract can thus be stated as follows:

Database coherence is preserved as long as the system guarantees the *N-A, N-I* and

N-D properties for all transactions, while each programmer ensures the N-C property for his transaction trees (figure 5).

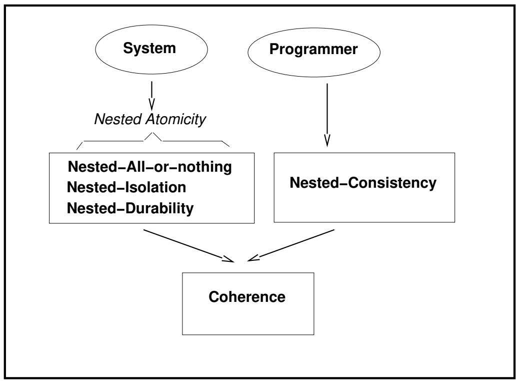

Figure 5: The new coherence contract

From a programmer point of view, this contract transforms the problem of maintaining the coherence of a concurrent nested transactional database system, into the problem of maintaining the coherence of sequential execution of transaction trees. Consider for instance, the transaction tree $\{T1, T11, T12\}$ (figure 2). Assume that neither $T11$ nor $T12$ is a necessary child transaction of $T1$, i.e. $N_{T1} = \emptyset$, and that requests are ordered with each other and with child transactions, e.g. $O_T1(req11, req12)$, $O_T1(req11, T11)$, $O_T1(req12, T11)$, $O_T1(req11, T12)$, $O_T1(req12, T2)$ and $O_T11(req111, req112)$. To ensure *nested-consistency*, one must guarantee that each sequential execution in figure 6-(a), when started from coherent objects' states, return coherent replies and coherent final states.

It is possible to reduce the number of cases to be considered by either preventing subtransactions to perform concurrently, or designing necessary subtransactions. For example, one can force $T11$ to execute before $T12$ (by stating that $O_{T1}(T11, T12)$), and design $T11$ as a necessary subtransaction of $T1$. In this case, the executions to be considered will be those of figure 6-(b).

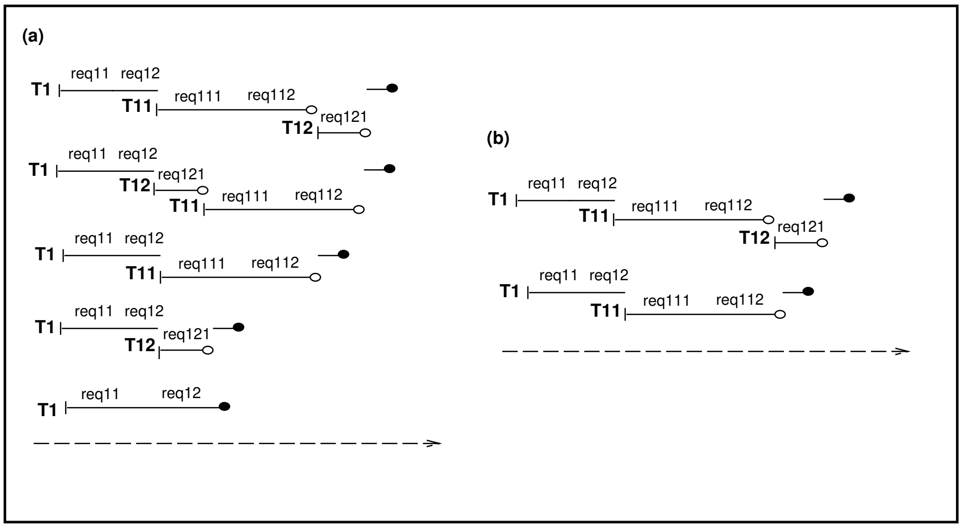

Figure 6: Sequential executions

5. CONCLUDING REMARKS

In this paper, we have presented a precise semantic of nested transactions by defining their *N-ACID* properties: *(N-A) nested-all-or-nothing*, *(N-C) nested-consistency*, *(N-I) nested-isolation* and *(N-D) nested-durability*. These properties permit to redefine the coherence contract for nested transactions: coherence is preserved as long as the database system guarantees the *N-A, N-I*, and *N-D* properties for all transactions, while programmers ensure the *N-C* property for their transaction trees.

Compared to the original coherence contract, the new one is more difficult to achieve from the programmer point of view. Programmer must deal with transaction tree executions instead of individual transactions. The increase of complexity is the price to pay for the flexibility and efficiency gained by the use of a nested transactions. A future work would lead to determine how the underlying database system might ensure the *N-C* property, assuming a minimal knowledge about coherence from programmers.

The nested model considered in this paper does not make any limitations neither on objects accesses, nor on parent/child concurrency. It encompasses other nested transactions models which prevent either one or both of these features [2,3,9,10,16]. Therefore, our coherence contract may be successfully applied to all those models.

ACKNOWLEDGMENTS: I am very grateful to Jean Ferrié and Bill Weihl for their helpful comments.

[1] Bernstein A.J, Hadzilacos V and N. Goodman. *Concurrency Control and Recovery in Distributed Database Systems*. Addison Wesley. 1987.

[2] Bernstein A.J and Lewis P.M. *Concurrency in Programming and Database Systems*. Jones and Bartlett, 1993.

[3] Eppinger J.L, Mummert L.B and Spector A.Z. *Camelot and Avalon: A Distributed Transaction Facility*. Morgan-Kaufman. 1993.

[4] Eswaran K.P, Gray J.N, Lorie R.A and Traiger I.L. "The notion of consistency and predicate locks in a database system". Communications of the ACM, Vol 19, Num 11. 1976.

[5] Guerraoui R, Capobianchi R, Lanusse A and Roux R. "Nesting Actions through Asynchronous Message Passing: the ACS protocol". Proceedings of the European Conference on Object Oriented Programming. LNCS, Springer Verlag, pp 170.184. 1992.

[6] Guerraoui R. "Towards Modular Concurrency Control for Distributed Object Oriented Systems". IEEE Proceedings of the International Workshop on Future Trends in Distributed Computing Systems, pp 240.247. 1993.

[7] Guerraoui R. "Atomic Object Composition". Proceedings of the European Conference on Object Oriented Programming. LNCS, Springer Verlag, pp 118.138. 1992.

[8] Harder T and Reuter A. "Principles of Transaction-Oriented Database Recovery". ACM Computing Surveys, Vol 15, num 4. 1983.

[9] Lynch N, Merrit M, Weihl W and Fekete A . *Atomic Transactions*. Morgan-Kaufman. 1993.

[10] Liskov B and Sheifler R. "Guardians and Actions: Linguistic Support for Robust, Distributed Programs". ACM Transactions on Programming Languages and Systems, Vol 5, Num 3. 1983.

[11] Moss J.E.B. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, MA. 1985.

[12] Papadimitriou C.H. *The Theory of Database Concurrency Control*. Computer Science Press. 1987.

[13] Reed D.P. *Naming and Synchronization in a Decentralized Computer System*. Ph.D Thesis, MIT. 1978.

[14] Taylor D.J. "How big can an atomic action be ?". IEEE Procedings of the Symposium on Reliability in Distributed Software and Database Systems. 1981.

[15] Weihl W.E. "Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types". ACM Transactions on Programming Languages and Systems, Vol 11, Num 2. 1989.

[16] Weikum G. "Principles and realization strategies of multilevel transaction management". ACM Transactions on Database Systems, Num 16, pp 132.180, 1991.