UNIVERSITY OF CALIFORNIA
Santa Barbara

Synchronization in Nested Transactions

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy
in
Computer Science
by

Rodolfo Ferreira Resende

Committee in Charge:

Professor Amr El Abbadi, Chairperson

Professor Divyakant Agrawal

Professor Oscar Ibarra

April 1994

The dissertation of Rodolfo Ferreira Resende
is approved:

Committee Chairperson

April 1994

April, 1994

Acknowledgements

This dissertation exists because of the help and comfort from many sources. I can hope to mention only a few important people. To all the others not here please accept my thanks.

I would like to thank my advisor Professor Amr El Abbadi, for his guidance and assistance during the development of the ideas presented in this dissertation. It was a great honor to work with such a brilliant and admirable scholar. I was fortunate to have the opportunity of working with Professor Divyakant Agrawal. His advice and encouragement were invaluable. I am grateful to Professor Oscar Ibarra for serving on my committee.

I appreciate the help and friendship of all the members and ex-members of the Distributed Systems Laboratory.

My parents and in-laws were always a source of encouragement and strength. They deserve special thanks for just being who they are.

Delia, my wife, and Leonardo, my son, helped me through the difficult phases of this research. Without their help, I probably would never have made it.

VITA

January 12, 1959 —	Born in Belo Horizonte, MG Brazil
1977-1981 —	B.Sc., Universidade Federal de Minas Gerais
1982-1986 —	M.Sc., Universidade Federal de Minas Gerais
1982-1989 —	Systems Analyst, UFMG
1989-1994 —	Ph.D., University of California, Santa Barbara
April, 1994 —	Ph.D., University of California, Santa Barbara

PUBLICATIONS

- Rodolfo F. Resende and Amr El Abbadi. *A Graph Testing Concurrency Control Protocol for Object Bases*, Proceedings of the 4th International Conference on Computing and Information, May 1992, 316-319.
- Rodolfo F. Resende and Amr El Abbadi. *On The Serializability Theorem for Nested Transactions*, Technical Report TRCS 93-20, University of California at Santa Barbara, 1993. To appear in Information Processing Letters.
- Rodolfo F. Resende, Divyakant Agrawal, and Amr El Abbadi. *Semantic Locking in Object-Oriented Database Systems*, Technical Report TRCS 94-01, University of California at Santa Barbara, 1994.

FIELDS OF STUDY

- Concurrency Control in Database Systems
- Distributed Systems

ABSTRACT

Synchronization in Nested Transactions

by

Rodolfo Ferreira Resende

Database systems take advantage of concurrent activities in order to offer better throughput and response time for programs accessing the shared data. In spite of employing concurrency, database systems offer a a programming environment where the applications programmer is not required to write programs that have to synchronize to each other. The scheduler is the database system component that manages the concurrent accesses. The execution of application programs as seen by the scheduler is called a transaction. The scheduler ensures that, in spite of the possible interleaving of accesses, the effects of the transactions will appear atomic to each other. The scheduler does this by following a concurrency control protocol. Transaction atomicity in term of concurrency control issues is know as *serializability*. Most of the schedulers ensures conflict serializability, by basing their decisions on a conflict or commutativity based predicate defined on the atomic operations invoked by the transactions. Conflict and commutativity predicates capture the notion of a potential interference between transactions. Traditional serializability theory deals with "flat" transactions, i.e., transactions that can only call atomic database operations.

This dissertation presents two contributions in the area of serializability of a special type of transaction called nested transactions. Nested transactions mainly differ from traditional or "flat" transactions in the sense that they offer concurrency control within transactions by serializing subtransactions, i.e., transactions can call other transactions. There exists two components in a nested transaction execution. The first component is the hierarchy of transactions and the second is the level of the atomic database accesses operations. Our first contribution is a theoretical framework where the serializability of nested transactions can be formally proven. Such framework can be used whenever serializability is based on a conflict predicate at the level of the atomic database access operations. The second contribution is the specification of protocols that can be used in schedulers coping with nested transactions. The protocols are novel in the sense that they can use semantic information of the programs associated to the hierarchy of transactions in order to enhance concurrency. We present formal correctness proofs of

the proposed protocols.

Contents

List of Figures

Chapter 1

Introduction

A database consists of a collection of objects. A database state maps the database objects into object values. Some of the database states are assumed to be *consistent*. The state of the database reflects the current state of the part of the real world being modeled through the database. Not all possible combinations of values represent a legal state of the world. This is translated in database systems by the notion of *integrity constraints*. States that obey integrity constraints are called *consistent states*.

In order for the database to evolve and follow the part of the world that it represents, it is necessary to execute application programs. Some application programs change the state of the database and others simply read the current state. The objects of the database are *shared* by the application programs. In order to maintain the consistency of a database we have to require that the executions of the application programs *preserve database consistency*. That is, whenever a program execution starts with a database state that is consistent, it must leave the database in a consistent state after it terminates. Consistency preservation captures the concept of producing database states that are meaningful [BHG87].

A computer system may want to support more than one program executing at the same time, i.e., to support *multiprogramming*. This is done in order to benefit from the potential parallelism of processor and I/O activity. The superposition of processor and I/O activity provides efficiency. A database can also be distributed in terms of data and processing among two or more computer systems. Such distribution also generates concurrency problems similar to the ones caused by multiprogramming. In such concurrent systems the execution of database application programs gives origin to potential interleaving and interference of database accesses.

It is well known that the design of programs that have to synchronize with other programs is more difficult than the design of programs where there is no such need. For this reason we want to give the application programmer the flexibility of designing programs that are unaware of each other. If we assume that each such a program produces correct executions then a sequence of program executions will also be correct. Here what we mean by correct execution is an execution that preserves database consistency. However in a concurrent system, due to the interleaving of accesses, an execution that would be correct if the program was running alone, can result in an inconsistent database [Pap86, BHG87]. This is due to the non-controlled interference of operations over the shared data.

This gives rise to two requirements:

- We want the database system to take advantage of multiprogrammed execution of application programs in order to have efficient system utilization.
- We do not want that the application programmer must design programs that have to synchronize with other programs, because this makes the task of programming more difficult.

These two requirements are satisfied with the adoption of a database component called *scheduler*. The *scheduler* is a program or collection of programs that controls the concurrent execution of (application) programs [BHG87]. It exercises this control by restricting the order in which the accesses to the shared data are executed. The scheduler bases its decisions by following some set of rules known as a *protocol*.

The interface between application programs and the scheduler gives origin to the concept of *transaction* [BHG87]. This is analogous to the concept of *process* at the interface of application programs and the operating system. Transactions add properties to the notion of process, we can informally consider that they are specialization of processes. It is possible that two or more processes form one transaction or that one process gives origin to two or more transactions. But usually one process correspond to one transaction.

The properties of a transaction depends on the system implementation. However one important property that most database systems implement is *transaction atomicity*. Transaction atomicity captures the notion that the transaction is executed either without any interference or none of a transaction's operations are performed [BHG87]. If transaction atomicity can be guaranteed then the two requirements above are satisfyed as will be discussed more formally later in this dissertation.

Among the operations at the interface of an application program and the scheduler an operation called *abortion* is usually provided. Because of failure or some other reasons that we do not specify further it may be necessary to abort a transaction. Transaction atomicity requires that if a transaction aborts either by program decision or because of some other reason then all its effects must disappear. In this dissertation we use the assumption that an execution involves only *committed*, i.e., non aborted transactions. *Concurrency control* relates to the synchronization of non-aborted transactions. *Recovery* relates to the issues of how to guarantee that aborted transactions do not interfere with the execution of committed transactions. We assume that there is a component of the database system, the *recovery manager*, that guarantees that all aborted transactions have their effects removed from an execution.

Transaction atomicity in terms of concurrency control translates into a concept called *serializability*. A *serial execution* is an execution composed by a sequence of non-interleaved transactions. We say that a scheduler ensures serializability if the executions produced by the scheduler have an *equivalent* serial execution of the involved transactions. In order to formalize these concepts, e.g., transactions, execution, equivalence, a model is used. In the models that we will adopt the scheduler is not described explicitly like for example it is possible using the model in [Hoa85]. Most of the research in the area of database concurrency control uses a model where the scheduler is not described explicitly, rather the scheduler is described by the properties of the executions allowed by the protocol that it follows. Such model is sometimes referred as the *classical model* [FLMW90].

A model deals with at least the following three entities:

- The shared data. Access to the data is provided by atomic operations that return a value and operate a state transition.

- Application programs that access the shared data. The execution of one such program is called a *transaction*.

- The *scheduler*, responsible for the scheduling of the transactions accesses to the shared data.

The necessity of a model comes from the complexity of putting together all the elements involved in such concurrent system, and proving that a certain protocol ensures serializability.

In a database system, beyond the operations related to shared data access there are the *transaction operations*. Transaction operations are defined at the interface of the application programs and the scheduler. Usual transaction operations are start a transaction, abort a transaction, commit a transaction. The transaction operations supported by a system give origin to basically three different categories of transactions as follows.

1. Flat transactions. Such transactions occur in systems that offer to the programs an operation to start a a transaction, and then, the program can only issue access operations or terminate the transaction. This establishes two levels: the level of the transactions and the level of the transaction and database operations. If one considers only the level of the transactions, such approach renders a single-level transaction system, hence the denomination “flat”.

2. Multilevel transactions. Such transactions occur in systems that offer to the programs the same type of operations to start and to terminate a transaction as before, but there exists two or more levels of transactions and database access operations [Wei91]. One operation that appears atomic at one level is implemented as a transaction, with operations in the immediately next lower level.

3. Nested transactions. In this case the system offers to the programming system an operation to start a transaction, and after this, the program can arbitrarily start transactions within other transactions, i.e., subtransactions, in a way that resembles subroutine calls. The caller-callee relationship induced by such calls generate a hierarchical structure, and hence the name nested transactions [Mos85].

One nice aspect of flat transactions is the simplicity of their structure. Multilevel transactions are used basically because of the layered abstraction that they allow. Below we present some motivations for the use of nested transactions.

- If a program organizes the accesses to the shared data using nested transactions the abortion of subtransactions can be made in an independent way. This must be considered in contrast with flat transactions where abortion usually implies losing everything that was done in the whole transaction unit unless some special mechanism is adopted. Such aspect of nested transactions is also known as *intra-transaction recovery* [HR93].

- If the database is distributed then nested transactions provide a mechanism to distribute activities in different sites.

- They provide synchronization between concurrently running parts of the same nested transactions, i.e., they can be used to express executions that can be carried in a concurrent way. This aspect is also known as *intra-transaction parallelism* [HR93].

- In object oriented databases if the execution of a method is seen as a transaction then the execution of methods generates a hierarchy that maps in a convenient way into nested transactions [HH91].

- Nested transactions gives a great flexibility in terms of safe composition of application programs whose modules may be designed and implemented independently. One important aspect of this modularity is that we can associate semantics to each module or subtransaction.

The results of the classical theory of database concurrency control are basically related to the flat transactions. The theory defines the notion of *conflict*. Informally conflict is a relation on database operations that captures the concept of whether an operation can interfere with another operation. There are basically two approaches for defining conflicts. One approach is to enumerate the operations of a system and to define the conflict relation explicitly [BHG87]. Another way of defining a conflict relation is to specify the conditions for conflict and leave the operations unspecified [Kor83]. An execution is conflict serializable if there exists a serial execution of the same transactions where the order of the conflicting pairs is the same in both. The theory prescribes how to construct a graph that represents the execution of transactions in a database system. Such a graph has nodes representing the transactions and edges representing the existence of operations that *conflict*. The fundamental theorem of the theory relates the conflict serializability of an execution with the absence of cycles in its corresponding graph [BHG87]. Most of the results of the classical theory of database are also available in terms of multilevel transactions [Wei91].

Our first contribution presented in this dissertation is related to extending the results of conflict serializability of the classical theory from flat to nested transactions. We have developed the definitions and proofs for a theorem applicable to nested transactions that emulates the fundamental theorem of serializability for flat transactions.

We can assume that each subtransaction in a nested transaction corresponds to the occurrence of a certain operation defined by the corresponding program. There are occasions where conflicts that occur in the lower levels of an execution can be ignored because of semantic information associated to the subtransactions in the higher levels. Thus this semantic information can be used to improve concurrency. We use the term *conflict serializability* to refer to the serializability that considers all the conflicts in an execution, whereas *semantic serializability* is used to refer to the serializability that allows certain conflicts to be ignored because of semantics.

A second contribution that we present in this dissertation is the design of protocols for concurrency control of nested transactions that can use semantic information. Hence we abandon conflict serializability as correctness criterion and we adopt semantic serializability for proving the correctness of our protocols. We have adopted the model of Beeri, Bernstein and Goodman [BBG89] as the framework for discussing the correctness of such executions. Its main advantage is that it is an extension of the classical theory, with extensions to deal with the added complexity of nested transactions.

This thesis is divided into six chapters. In Chapter 2 we present a summary of the model of the system that supports nested transactions. In Chapter 3 we extend the correctness criteria of the classical theory by presenting a definition of conflict serializability for concurrency control of nested transactions. We define a serialization graph for nested transactions and we prove that its acyclicity is a sufficient and necessary condition for conflict serializability. In Chapter 4 we introduce the motivations for the use of semantics in enhancing concurrency. We present protocols for concurrency control of nested transactions that can take advantage of semantic information in Chapter 5. The presentation of these protocols is done in the intuitive scenario created by considering nested transactions as hierarchies of method executions of object oriented databases. We discuss related work along with these chapters. We conclude the dissertation with a summary of our results and a discussion of future work.

Chapter 2

A Model for Nested Transactions

2.1 Introduction

In this chapter we present a model for systems that employ nested transactions. The model is essentially the same as developed by Beeri, Bernstein and Goodman [BBG89]. This model extends the classical model of Bernstein, Hadzilacos and Goodman [BHG87] by making explicit use of the concepts of *states* and *return values*, and by adding a hierarchical structure to the transaction component of an execution.

In Chapter 3 we will make use of only a few features of the model. It is possible to develop a model of concurrency control of nested transactions that is a straightforward extension of the classical model. It is possible to describe our results in a simpler model without the complexity of the model we adopt here. But we need a more complete model to justify formal aspects used in Chapter 4. To avoid the complexity of dealing with two models we use elements of the more elaborate model, i.e., [BBG89], even when dealing with conflict serializability of nested transactions.

The model basically defines the modeled components, and describes through *axioms* what is expected from executions or computations. Executions that obey these axioms are suitable to be used in equivalence-preserving transformations described later. Before we present the nested transaction model, we will present a summary of serializability theory for flat transactions [BHG87].

2.2 Flat transactions model

The interface of an application program and the scheduler gives origin to the concept of transaction. A transaction corresponds to the trace of execution of an application program as seen by the database. The key property of a transaction is atomicity. Sometimes the atomicity of a transaction, understood as an indivisible unit, is referred as *isolation*.

A transaction can either *commit* or *abort*. When a transaction commits we also requires that its state transformation must take effect in the database, this is sometimes called *durability* or *persistency*. As part of the concept of atomicity we want that an aborted transaction does not affect the database. Concurrency control consists of the issues related to the synchronization of committed transactions. Recovery consists of the issues related to the guarantee that operations of aborted transactions do not affect the database. As already mentioned, in this dissertation we assume that there exists some recovery mechanism that guarantees that aborted transactions do not affect the database. We discuss concurrency control issues. Serializability is the counterpart of transaction atomicity in terms of concurrency control. Serializability theory is a mathematical tool that allow us to prove whether or not a scheduler works correctly [BHG87].

We begin considering objects and operations. The shared data consists of a set of objects and each object supports certain operations. In some models only *read* and *write* operations are considered. These are referred to as *read/write* models. When only read and write operations are considered the model is usually simpler than models that consider arbitrary operations.

A model captures the notion that one operation can interfere with another using a *conflict* or *commutativity* relationship. It is possible that a model does not capture the notions of database state transformation or values returned by operations. In this latter case we are dealing with *uninterpreted* features. However we can use such notions to derive properties of elements of the model. A pair of operations is said to *conflict* if when considered in isolation the order of execution may affect: (a) the results returned by one of them, or (b) the final value of the objects of the database. Using this criterion it is easy to understand why in read/write models there exists a simple rule: two operations conflict if both are on the same object and at least one of them is a write operation. The model we will adopt for nested transactions interprets the notion of state transformation and return values. In this latter model we have two formal definition of *commutativity* one that is state dependent and the other state independent.

Transactions can be described as sequences of operations [Pap86], or as a partial order of operations [BHG87]. The advantage of modeling transactions as a partial order is that the transaction need not specify the ordering of every two operations that appear in it. Beyond being a sequence or a partial order of operations some conditions must be satisfied. Lets review the formal definition of a transaction in [BHG87]. A *transaction* T_i is a partial order with ordering relation $<_i$ where

1. $T_i \subseteq \{r_i[x], w_i[x] \mid x \text{ is a data item}\} \cup \{a_i, c_i\}$;

2. $a_i \in T_i$ iff $c_i \notin T_i$;

3. if t is c_i or a_i (whichever is in T_i), for any other operation $p \in T_i$, $p <_i t$; and

4. if $r_i[x], w_i[x] \in T_i$, then either $r_i[x] <_i w_i[x]$ or $w_i[x] <_i r_i[x]$.

Condition (1) defines the operations of their model. The model in [BHG87] is a read/write model. The operations a_i and c_i correspond to *abortion* and *commit* respectively. When dealing with an aborted transaction one should consider that its operations have no effects in the database. A committed transaction is a transaction that successfully operates a database state transformation. Condition (2) says that a transaction either aborts or commits. Condition (3) says that the abortion or commitment of a transaction follows all the other operations. Condition (4) says that it is the responsibility of the transaction to order the conflicting operations.

Commit and abort operations are included in [BHG87] because they also discuss recovery in their work. Since we do not discuss recovery issues in this dissertation we will not emphasize such operations. We will assume that we are only dealing with committed transactions. Observe that if we assume that we are dealing with committed transactions only conditions (1) and (4) are relevant.

When a set of programs generating transactions execute concurrently, their operations may be interleaved. Such a concurrent execution of transactions is called a *history*, or *computation*, or *schedule*, or simply an *execution*. A history indicates the order in which the operations of transactions were executed relative to each other. Similarly to the case of transactions, histories can be described as sequence of operations or as a partial order of operations. Some well-formedness criteria has to be added. Lets now review the concept of history in [BHG87]. Let $T = \{T_1, \ldots, T_n\}$ be a set of transactions. A *history* H over T is a partial order with ordering relation $<_H$ where:

1. $H = \bigcup_{i=1}^{n} T_i$;

2. $<_H \supseteq \bigcup_{n}^{i=1} <_i$; and

3. for any two conflicting operations $p, q \in H$, either $p <_H q$ or $q <_H p$.

Condition (1) says that the operations of H are the operations of the involved transactions. Condition (2) says that H must honor the orderings of the involved transactions Condition (3) says that all pairs of conflicting operations are ordered. Since [BHG87] wants to distinguish between a prefix and a complete history, the concept above is actually called in their work a *complete history*. We have presented the model of [BHG87]. We now would like to take advantage of this presentation by also summarizing their correctness theory.

Recall that we want to be able to characterize atomic transactions. In terms of concurrency control this translates in serializable histories. Intuitively a history is serializable if it has the same effects of a history where transactions are executed one after the other, i.e., a serial history. Lets review how this is presented in [BHG87].

Two histories H and H' are equivalent ($\equiv$) if

1. they are defined over the same set of transactions and have the same operations; and

2. they order conflicting operations of nonaborted transactions in the same way; that is, for any conflicting operations p_i and q_j belonging to transactions T_i and T_j respectively where $a_i, a_j \in H$, if $p_i <_H q_j$ then $p_i <_{H'} q_j$.

As [BHG87] remark, the idea underlying this definition is that the outcome of a concurrent execution of transactions depends only on the relative ordering of conflicting operations.

A history H is *serial* if, for every two transactions T_i and T_j that appear in H, either all operations of T_i appear before all operations of T_j or vice versa.

A history H is serializable if it is equivalent to a serial history H_s. Since we are not dealing with prefixes of histories as in [BHG87], we do not have to care about them.

The determination of whether a history is serializable can be done by analyzing a graph derived from the history called a serialization graph. The *serialization graph* (SG) for H, denoted $SG(H)$, is a directed graph whose nodes are the committed transactions in H and whose edges are all $T_i \rightarrow T_j$ $(i \neq j)$ such that one of T_i's operations precedes and conflicts with one of T_j's operations in H.

The fundamental theorem of serializability theory says that a history H is serializable iff $SG(H)$ is acyclic [BHG87]. Recall that this result relates to flat transactions, our first contribution in this dissertation was to establish a similar theorem for nested transactions. In the next section we give an informal description of how the flat transaction model of [BHG87] extends to the nested transaction model of [BBG89].

2.3 From flat to nested transactions

Beeri, Bernstein and Goodman [BBG89] describe a model for nested transactions. We based our research in this model. In this section we discuss some of the important differences between flat and nested transactions model. The model of Hadzilacos and Hadzilacos [HH91] is not a "pure" model in nested transactions since they described it in the context of object oriented databases, where transactions are method executions. We will discuss their model trying to abstract from this specialization.

The first key difference between nested and flat transaction models is the fact that the transaction component of the execution is more complex. In flat transactions models most of the complexity is in the level of the atomic operations, and the transaction component is quite simple. In the case of nested transactions the transaction component is more complex. In the nested transaction model of [BBG89] the level of the atomic operations is called a DM. This is to evoke that such level is an execution performed by the *Data Manager* component of a database. The Data Manager is the program or collection of database programs that execute the operations. The component of an execution that is related to transactions is now a hierarchical structure. One interesting insight of [BBG89] is how to operate or transform these two components, the hierarchical transactions and the DM, producing a new execution in a way that preserves equivalence. We will explore such transformations in later chapters of this dissertation.

The work of Bernstein, Hadzilacos and Goodman [BHG87] has some *implicit* assumptions about the computer and database systems that are explicit in [BBG89]. The work of [HH91] also has some implicit assumptions. In [BBG89] eight *axioms* are listed that describe how a system is expected to behave. There is a reasonable number of definitions connected to these axioms. We adopt all these axioms since we think that systems that obey such axioms are in a certain sense intuitive and well behaved.

Another difference between [BHG87] and [BBG89] is that the latter deals explicitly with the notion of database states and return values of operations in the model. Recall that in the read/write model of [BHG87] such notions are not elements of the model.

In [BHG87] it is required that all pairs of conflicting operations, in a history, are ordered. Moreover the notion of equivalence says that in an execution it is only the ordering of conflicting operations that matters. In [BBG89] this issue is axiomatized with the concept of *Order Preserving Atomicity* (OPA). They use partial orders in the definitions of transactions and executions. OPA is used to axiomatize that the partial order at the level of the operations (the DM level) is equivalent to some totally ordered sequence of the involved operations.

In [BBG89] and [HH91] a transaction is defined as a partial order of operations. Hence a transaction is a flat structure. In [HH91] there exists a special object (environment) that can start transactions, such transactions are called *top-level* transactions. In [BBG89] a transaction can be started by a non specified object, and in this case it is a top-level transaction, or it can be started by another transaction (its parent) and in this case it is called a subtransaction. A subtransaction is viewed by its parent as an operation. They assume that there exists a convenient mapping between operations and transactions.

In the next section we formally summarize the model of [BBG89].

2.4 Nested transactions model

The components of a computation are states, operations, transactions, return values, and a partial order. Each state represents the state of the entire database, states are denoted by $St = \{s_1, s_2, \ldots\}$. The set of transactions are denoted by $T = \{T_1, T_2, \ldots\}$. The set of operation occurrences in computations is denoted by OP.

A *DM* or *Data Manager* represents the storage subsystem. The DM is characterized by the set of computations it can execute. A computation is described by a tuple. A

DM tuple is a tuple $(s_1, O, <, r, s_2)$, where s_1 and s_2 are states, O is a finite subset of OP, $<$ is a partial order on O, and r is a return value function with domain O. A *D-computation* is a DM tuple that obeys axioms (C1), (C2), and (C3) below. A *DM* D is the set of all D-computations of a given system. A DM tuple $c = (s_1, O, <, r, s_2)$ in D means that starting from state s_1, D can execute the operations in O in the execution order $<$, return the values specified by r, and end in state s_2. For $o_1, o_2 \in O$, we say that o_1 *precedes* o_2 in c if $o_1 < o_2$. To denote that o_1 does not precede o_2 we write $o_1 \not< o_2$. If $o_1 \not< o_2$ and $o_2 \not< o_1$ then o_1 and o_2 are *concurrent*. A D-computation $(s_1, O, <, r, s_2)$ is *serial* if $<$ is a total order on O. A D-computation $c = (s_1, O, <, r, s_2)$ is *order preserving atomic (OPA)* if there exists a total order $<_1$ such that $< \subseteq <_1$ and $\hat{c} = (s_1, O, <_1, r, s_2)$ is also a D-computation. That is, c has the same effect as a serial computation that preserves the order of c's operations. Let $O_1, O_2 \subseteq$ OP, such that $O_1 \cap O_2 = \emptyset$, let $<_1$ and $<_2$ be partial orders on O_1 and O_2 respectively, and let $<_1 \otimes <_2$ be defined as $<_1 \cup <_2 \cup O_1 \times O_2$. Square brackets are used to restrict domains, e.g., $r[O]$ denotes function r restricted to domain O. A DM D must satisfy:

(C1) (Composition axiom) If, for some s_1, s_2, s_3, the computations

$$c_1 = (s_1, O_1, <_1, r_1, s_2) \text{ and } c_2 = (s_2, O_2, <_2, r_2, s_3)$$

are in D then so is their *composition*

$$c = c_1 \otimes c_2 = (s_1, O_1 \cup O_2, <_1 \otimes <_2, r_1 \cup r_2, s_3)$$

(C2) (Decomposition axiom) If, for some s_1 and s_3, there is a computation

$$c = (s_1, O_1 \cup O_2, <_1 \otimes <_2, r, s_3) \text{ in } D,$$

then there exists a state s_2 in St such that

$$c_1 = (s_1, O_1, <[O_1], r[O_1], s_2) \text{ and } c_2 = (s_2, O_2, <[O_2], r[O_2], s_3) \text{ are in } D.$$

(C3) All D-computations are OPA.

Axioms (C1) and (C2) show how to compose and decompose tuples. Axiom (C3) entails the assumptions in the classical theory of serializability that operations are atomic and that noncommutative operations are related by $<$.

A transaction is an execution of a program. Formally, a *transaction* is a tuple $t = (O, <_t, r)$, where O is the set of operations executed in the computation, $<_t$ the *transaction order*, is a partial order on O, and r is a return value function with domain $O \cup \{t\}$. A *transaction tuple* for t, or a t-tuple, is any tuple $(O, <, r)$. A *t-computation* is a t-tuple that satisfies axiom (C4):

(C4) (Order extension axiom) If $t = (O, <_t, r)$ is a transaction, then $(O, <, r)$ is a t-computation if and only if $<_t \subseteq <$.

A subtransaction is viewed by its parent as an operation, though in reality it is an execution of a program. To achieve this dual view, we assume a function from the set T of transactions to a subset OP_T of OP, which gives the operation of which the (sub)transaction is an implementation. As an abuse of functional notation $tr(o)$ denotes any transaction associated with operation o. Axiom (C5) below states that $tr(o)$'s effect can be achieved by o, i.e., it relates an operation's specification to its implementation.

(C5) (Sequential correctness axiom) Let $tr(o)$ be a transaction associated with operation o. For all s_1, s_2, $<$, r, if $(O, <, r)$ is a $tr(o)$-computation, and if

$$(s_1, O, <, r[O], s_2)$$

is a serial D-computation, then

$$(s_1, \{o\}, \emptyset, r[tr(o)], s_2)$$

is also a D-computation.

A *hierarchical tuple* is a tuple $c = (s_1, F, <, r, s_2)$ where s_1 and s_2 are initial and final states respectively. F is a forest of subtransactions and operations, $<$ is an execution order, and r is a return value function. The special case in which F is a one-level forest, c is also a DM tuple. Sometimes it is convenient to treat a given forest as a tree. This can be done by adding a root (in [FLW90] this root is called "mythical" T_0). Formally, we assume that a hierarchical tuple $c = (s_1, F, <, r, s_2)$ can also be represented by a $\hat{c} = (s_1, A, <, \hat{r}, s_2)$, where A is the same as F together with a distinguished transaction, T_0, that invokes as subtransactions the roots of F, and $\hat{r}$ is the same as r with domain extended to T_0, $r(T_0)$ is irrelevant.

The set of *computations forests* CFs [1] is the set of hierarchical tuples that satisfies axioms (C6)-(C8) below. For a hierarchical tuple c, the set of leaves is $leaves(A)$ or $leaves(c)$; the roots (top-level transactions) are $roots(c)$. For a node x, the subtree rooted at x is A_x, and its leaves are $leaves(x)$. The set of descendents of x is $desc(x)$ and the set of children (i.e. immediate descendents) is $child(x)$, ($roots(c)$ is the same as $child(T_0)$ or $child(A)$). The lowest common ancestor of nodes x and y is $lca(x, y)$. Nodes x and y are *incomparable* if neither is a descendent of the other. The *DM projection* of c is $c[leaves(c)]$, also denoted as $c[D]$. For an internal node x, *c restricted to x* is the transaction tuple $(child(x), <[child(x)], r[child(x) \cup \{x\}])$. A CF $c = (s_1, A, <, r, s_2)$ must satisfy:

(C6) (Downward compatibility) Let $x, y \in A$, and let $p \in desc(x) \cup \{x\}, q \in desc(y) \cup \{y\}$, if $x < y$, then $p < q$.

(C7) (Transaction validity) For each internal node x, c restricted to x is an x-computation.

(C8) (DM validity) The DM projection of c, $c[D]$, is a D-computation.

Axiom (C6) says that if x completes before y begins, then each operation and subtransaction of x completes before any operation or subtransaction of y begins. (C7) requires the consistency of $<$ with the partial order of each constituent transaction, and (C8) says that a CF comprises a computation of a DM D.

Proposition 2.1 *[BBG89] Let $c = (s_1, A, <, r, s_2)$ be a CF. The partial order $<$ can be extended to a partial order $<'$ such that $c' = (s_1, A, <', r, s_2)$ is also a CF and for all nodes x and y, if, for all $p \in child(x)$ and all $q \in child(y)$, $p <' q$ then $x <' y$.*

Proposition 2.1 establishes a "normalization" process for CFs. Such canonical partial orders will be assumed from now on, i.e., $<$ in the following CFs cannot be further extended as prescribed by Proposition 2.1. Furthermore, by DM-validity, (C8), for every CF c, $c[D]$ is a D-computation. By the OPA property of axiom (C3) we can extend the order to be serial on the leaves, and by Proposition 2.1, we can extend it further to be upward compatible. Hence we assume that all D-computations are serial.

[1] We will maintain the denomination "Computation Forest" or "CF", despite the introduction of T_0.

We have presented the model of [BBG89]. We now would like to take advantage of this presentation, like we did for flat transactions, by also summarizing their correctness theory.

In [BBG89] there are no graph based tool for serializability like in the case of flat transactions, they propose the use of transformations. In flat transactions if one wants to prove the correctness of a protocol the proof obligation is to show that for all the executions produced by such protocol their serialization graph is acyclic. The proposal of [BBG89] is that if one wants to prove the correctness of a protocol for nested transactions the proof obligation is to show that all executions produced by such protocol can be *transformed* in a certain form of serial sequence.

We will now present the definitions and transformations associated with such approach.

Two CFs c_1 and c_2 are *equivalent*, denoted $c_1 \equiv c_2$, if they have the same initial and final states and their roots have the same return values (which implies that they have the same roots).

A CF c is *serial* if $<$ is a total order on the roots and on the children of each node.

A CF c is *serializable* (SR) if it is equivalent to some serial CF.

We now use some definitions from [BBG89]. Let $<$ be a partial order. For sets X and Y, we write $X < Y$ if, for all $x \in X$ and $y \in Y$, $x < y$ holds. Let X be a set of nodes. We say that $<$ *separates* X or that X is *separated by* $<$, if for each $y \in leaves(A) - leaves(X)$, either $y < leaves(x)$ or $leaves(X) < y$.

Let $d = (\hat{s}_1, T_d, <_d, r_d, \hat{s}_2)$ be a computation tree, where $\hat{s}_1$, $\hat{s}_2$ are the initial and final states of $c_x = (\hat{s}_1, A_x, < [A_x], r[A_x], \hat{s}_2)$, T_d is a tree with root x, and $r_d(x) = r(x)$. Note that $d \equiv c_x$. The *substitution of* d *for* c_x *in* $c = (s_1, A, <, r, s_2)$, denoted $c[c_x \Leftarrow d]$, is

$$c[c_x \Leftarrow d] = (s_1, \tilde{A}, \tilde{<}, \tilde{r}, s_2)$$

where

1. $\tilde{A}$ is A (from c) with A_x replaced by T_d.

2. $\tilde{<}$ equals $<$ on $A - A_x$ and $<_d$ on T_d. For $y \in A - A_x$, if $y < x$ (or $x < y$), then $y \tilde{<} T_d$ (or $T_d \tilde{<} y$).

3. $\tilde{r}$ is equal to r_d on T_d and is equal to r elsewhere.

Substitution Theorem. Let c be a CF, x a node separated by $<$, and d a computation tree with root x such that $d \equiv c_x$. Then $\tilde{c} = c[c_x \Leftarrow d]$ is a CF and $c \equiv \tilde{c}$.

The substitution theorem of [BBG89] is important in the sense that they put their correctness theory apart from the correctness theories that establish equivalence of the *same* system. Consider for example the flat transactions correctness theory, there we establish equivalence between histories involving the *same* operations. It is basically the substitution theorem that gives the correctness theory of [BBG89] its power and also its complexity.

The substitution theorem can be specialized in *reduction* and *expansion*. We say that a substitution is a reduction when the substituted node is not only separated as required in the above theorem but also all its children are leaves and the node that is replacing it is a leave. Expansion is the reverse of reduction, the substituted node is a leave and all the children of the node that is replacing it are leaves.

We now present the definition of commutativity presented in [BBG89]. Let o_1 and o_2 be operations and s be a state. We say that o_1 and o_2 *commute* with respect to s if for all r and $\bar{s}$, $(s, \{o_1, o_2\}, (o_1, o_2), r, \bar{s}) \in D$ iff $(s, \{o_1, o_2\}, (o_2, o_1), r, \bar{s}) \in D$. Two operations *(generally) commute* if they commute with respect to all states. Operation o_1 *conflicts* with o_2 if o_1 does not commute with o_2.

The definition of commutativity of [BBG89] can be specialized for example in *right backward commutativity* [Wei89]: We say that o_1 *right backwards commute* with o_2 if for all states s where $(s, \{o_1, o_2\}, (o_1, o_2), r, \bar{s}) \in D$ then $(s, \{o_1, o_2\}, (o_2, o_1), r, \bar{s}) \in D$. Relaxing the definitions of commutativity is one of the ways that enable a scheduler to accept more executions, we do not pursue this direction here. Observe that [BBG89] define commutativity only for operations. This may look restrictive but thanks to substitution, a node that is an internal node in one CF can correspond to a leaf in an equivalent CF, and then it may relate through commutativity to other operations, avoiding cautioning with respect to atomicity. In Chapter 4 we will introduce a definition of commutativity that involves internal nodes. We will adopt the same definition of commutativity as here for atomic operations but extended to handle commutativity between internal nodes.

We now continue our description of correctness in [BBG89] presenting more definitions that will enable the presentation of *Commutativity-based reversal* the second and last execution transformation.

Let $<$ be a partial order on A. The *pull down* of $<$, denoted $<\downarrow$, is defined by: $x <\downarrow y$ if for some ancestors p of x and q of y, $p < q$.

The relation

$$< tr = (\bigcup_{t \in A} <_t) \downarrow$$

is called the *transaction order* for c. Beeri, Bernstein and Goodman show that to determine if $x <_{tr} y$, simply observe the transaction order between the children of $lca(x, y)$ from which x and y are descendents. In later chapters we will refer to $<_{tr}$ as the *algorithmic order* of the execution.

Let x and y be adjacent leaves in a CF c. Since $\{x, y\}$ is separated, $c[D] = c_D^1 \otimes c_D^2 \otimes c_D^3$, where the operation of c_D^2 are x and y. Let $\hat{s}_1$ be the initial state of c_D^2. We say that *the order of x and y can be reversed in c* if they commute with respect to $\hat{s}_1$ and are not related by $<_{tr}$. In later chapters we will try to be less formal, informally what we mean here is that x and y can be reversed in an execution if they commute and they are not algorithmically related. In [BBG89] the reversal of two operations obtaining a new execution is formally presented including the changes in $<$.

Commutativity Theorem. Let x and y be leaves of c whose order can be reversed, and let $\tilde{c}$ be obtained from c by reversing their order. Then $\tilde{c}$ is a CF and $c \equiv \tilde{c}$.

The commutativity theorem is the formal basis of the Commutativity-based reversal transformation. This theorem allows the use of commutativity to change the order of adjacent leaves. Our results in Chapter 3 are all related to commutativity-based reversals.

Observe that [BBG89] define classes of computations that can be proved serializable using substitution and commutativity-based reversal. Classes of correctness are not distinguished here and a different nomenclature is adopted. A CF c is *semantically equivalent* to $\bar{c}$ if we can get $\bar{c}$ from c using substitution and commutativity-based reversal transformations. A CF is *semantically serializable* if it is *semantically equivalent* to some serial CF.

Observe that when a CF c is related to CF $\bar{c}$ by commutativity-based reversals, c and $\bar{c}$ have the same transactions and the same operations which means *same* system equivalence, We have adopted the term "semantic serializability" (as [MRW+93] also did) because when we use substitution then we have to consider that a certain node can be substituted by another. This implies that both nodes implements the same operation i.e. that they have the same "semantics".

2.5 Other models

We have mentioned that the model of [HH91] is similar to the model that we have adopted, despite the fact that it is specialized for object oriented databases.

In [HH91] a transaction corresponds to a method invocation. An operation in [BBG89] can correspond either to a *method step* or a *local step*. A local step corresponds to an operation executed by the DM whereas the method step is an operation that invokes a method execution, i.e., a transaction. In [HH91] the return values are tied directly to the local and message steps.

Most of the the axioms of [BBG89] have their correspondents in [HH91]. In [HH91] the concept of a well-defined execution description is described as a *legal history*. In [HH91] (C1) and (C2) are for free simply by using the mathematical concept of functions instead of the tuples of [BBG89]. Axioms (C3) and (C8) correspond to conditions 2-(b) and (3) of legal histories. Axioms (C4) and (C5) does not have an explicit correspondence. Axiom (C6) corresponds to condition 2-(c) of legal histories, and axiom (C7) corresponds to condition 2-(a) of legal histories.

The model of [FLW90] differs from the models that we have discussed more radically. They use the notion of *input/output automaton*. An input/output automaton describes events and state transformations of system components. Such formalism is very powerful but unfortunately more complex than the traditional theory. One notable difference is that they define serial executions in terms of the objects involved independent of a given interleaved execution. Such decoupling brings more generality (and complexity) to their model. In terms of correctness, in the next chapter we discuss how their result relates to ours.

Chapter 3

Conflict Serializability

3.1 Introduction

In this chapter we develop a notion of equivalence based on a conflict relationship of the atomic operations. Using this notion we present a definition of conflict serializability for concurrency control of nested transactions. Although our approach is more restrictive than [BBG89] it allows the development of a serializability theory similar to the classical theory. Our contribution is a group of definitions and proofs that enable proofs of correctness of protocols for nested transactions. We establish the correspondence between *serializability* and absence of cycles in a graph representing dependencies among transactions, in way similar to the classical theory for flat transactions.

3.2 Conflict consistency and conflict equivalence

In this section we define serializability for nested transactions based on the notion of conflicts. We then contrast this definition with other definitions.

We will adopt the same definition of commutativity as the one of [BBG89] also presented in Chapter model. Our definition of serial execution is also the same. But our definition of equivalence of executions is quite different.

Let $<$ and $<'$ be total orders such that $<$ is a permutation of $<'$. We say that $<'$ is *conflict consistent* [HH91] with $<$ if whenever o conflicts with and precedes o' in $<$ then o precedes o' in $<'$.

Two DM tuples $c = (s_1, O, <, r, s_2)$ and $c' = (s'_1, O', <', r', s'_2)$ are *conflict equivalent*, denoted $c \equiv c'$, iff $s_1 = s_1'$, $s_2 = s_2'$, $O = O'$, $<$ is conflict consistent with $<'$, and $r = r'$. Now we will prove that if two DM tuples are conflict equivalent and one of them is a D-computation then the other is also a D-computation. We will use the technique of reversing commutative operations presented in [BBG89] and [Pap86]. First we show that the DM-tuple obtained by inverting the order of two contiguous and commutative operations of a D-computation is also in D.

Lemma 3.1 *Let c be a DM tuple with operations x and y contiguous and commutative with respect to the state in which the first one was executed. Let c' be a DM tuple obtained from c by reversing x and y. If $c \in D$ then $c' \in D$ and $c \equiv c'$.*

Proof. Without loss of generality assume that $x < y$. Since $c \in D$, c obeys the decomposition axiom and can be represented as $c = c_1 \otimes (\hat{s_1}, \{x, y\}, (x, y), r[\{x, y\}], \hat{s_2}) \otimes c_2$. By definition of commutativity $(\hat{s_1}, \{x, y\}, (y, x), r[\{x, y\}], \hat{s_2}) \in D$ hence $c' = c_1 \otimes (\hat{s_1}, \{x, y\}, (y, x), r[\{x, y\}], \hat{s_2}) \otimes c_2 \in D$. That $c \equiv c'$ is obvious. □

Now we show the relation between conflict equivalence and membership in D.

Lemma 3.2 *Let $c = (s_1, O, <, r, s_2)$ and $c' = (s_1', O', <', r', s_2')$ be DM tuples. If $c \in D$ and c is conflict equivalent to c' then $c' \in D$.*

Proof. By induction on the number of pairs, k, of operations that are reversed in c with respect to c'. The basis (k=1) is trivial from Lemma 3.1. For the induction hypothesis assume that the lemma is true for $k-1$ pairs. Consider the pair of operations x and y that are reversed in c' and contiguous in c [1]. The operations of such a pair commute since the total order $<$ in c is conflict consistent with the order $<'$ in c'. Now we use Lemma 3.1 to obtain $\bar{c}$ which is in D, is conflict equivalent to c, and has only $k-1$ pairs of operations reversed with respect to c'. □

Two CFs $c = (s_1, A, <, r, s_2)$ and $c' = (s_1', A', <', r', s_2')$ are *conflict equivalent*, denoted $c_1 \equiv c_2$, iff $A = A'$, $r = r'$, and $c[D] \equiv c'[D]$. A CF $c = (s_1, A, <, r, s_2)$ is *serial* iff $<$ is a total order on the roots and on the children of each node. A CF is *conflict serializable* iff it is conflict equivalent to some serial CF.

[1] Such a pair exists, by choosing x and y to be the pair of operations that are reversed in c' and closest together in c (without loss of generality, suppose x precedes y in c). Suppose (to obtain a contradiction) that any operation z came between x and y in c. We have that x must precede z in c', since they are closer together in c than x and y; similarly z must precede y in c'. The combination of these two facts shows x precedes y in c', contradicting the definition.

Our notion of serializability differs from previously proposed notions. In [BBG89] two CFs are equivalent if they have the same initial and final states and their roots have the same return values. In [HH91] the equivalence concept was strengthened by requiring not only the same roots but also the same forest (with the same return value for each node). We strengthen the latter definition by requiring not only the same final state but also that the ordering relation ($<$) of one be conflict consistent with the ordering of the other, which we showed guarantees the same final state. We consider states and return values only to take advantage of the state-dependent commutativity definition, thus remaining within the confines of the general theory presented in [BBG89]. Note that in [FLW90] serial correctness for T_0 corresponds to serializability in [BBG89]. To illustrate the difference between the equivalence definitions of [BBG89] and [HH91], consider a transaction that increments an integer data item, using a read and a write operation. Consider two increment subtransactions belonging to two different top-level transactions. In [BBG89] the order of the two increments is irrelevant as long as the underlying read and write operations are not interleaved. In contrast, in [HH91], the order is significant since all operations must return the same values, including the read and write that implement the increment operation. The difference between our notion of equivalence and that of [HH91] can be easily illustrated by any view serializable execution of flat transactions that is not conflict serializable [BHG87]. Although more restrictive, our approach allows for the development of a serializability theory similar to the classical theory for flat transactions. In the next section we develop a serialization graph for nested transactions and show that the acyclicity of the graph is both necessary and sufficient for conflict serializability. Hence, for the subset of executions that strictly depends on conflict information from the leaves, our approach complements the theory of Beeri, Bernstein and Goodman [BBG89]. In particular correctness in that theory is based on operational techniques such as commutativity-based reversals and substitution while conflict serializability can be proven using graph-theoretical techniques.

3.3 The Serializability Theorem

In this section we prove a theorem which establishes a necessary and sufficient condition for conflict serializability of nested transactions, similar to the fundamental theorem of serializability [BHG87] for flat transactions. First we define the serialization graph for nested transactions then we present the theorem.

The *serialization graph* of a CF $c = (s_1, A, <, r, s_2)$, $SG(c)$, is a directed graph whose nodes correspond to the nodes A in c and which has an edge $x \to y$ iff x and y are siblings and either

(a) There exist leaves x' and y' descendents of x and y respectively, so that x' precedes and conflicts with y'; or

(b) $x <_t y$ where t is the parent of x and y.

In our model once two operations conflict, this conflict propagates up to the level of the children of their least common ancestor which are their ancestors. Type (a) edges capture the order of conflicting operations. Edges of type (b) correspond to the transaction order or algorithmic precedence relation between siblings. We first prove a technical lemma, then we show that the acyclicity of our graph is both necessary and sufficient for conflict serializability. Our sufficiency part of the proof follows the proof in [HH91].

Lemma 3.3 *Given a graph $G = (V, E)$, a set of vertices $C \subseteq V, |C| \geq 2$, and P, a set of two or more disjoint nonempty partitions of vertices in C. Consider a graph $G' = (P, E')$ where $p \to p' \in E'$ iff there exists $v \to v' \in E$ and $v \in p, v' \in p'$. If C represents vertices which are involved in a (simple) cycle in G then there is at least one (simple) cycle in G'.*

Proof. Let f map $v \in C$ to its partition $f(v) \in P$. Consider the cycle of C's vertices $v_1 \to v_2 \to \cdots \to v_n \to v_1$. If f is applied to each element v_i of the cycle the result after removing possible self-cycles is a cycle between nodes of G'. Since there are two or more partitions, there is at least one simple cycle among them. □

Theorem 3.1 *(The Serializability Theorem) Let $c = (s_1, A, <, r, s_2)$ be a CF, c is conflict serializable iff $SG(c)$ is acyclic.*

Proof. (if) We will show how to construct c_s a CF that is serial and conflict equivalent to c.
[Definition of $<_s$] Let the level of a node in $SG(c)$ be the number of its proper ancestors in A (thus *root*(A) is at level 0). $<_s$ is the relation at the end of the execution of the following Hierarchical Topological Sort (HTS) [HH91]:

$<_s := \{(x,y) :$ there is an edge $x \rightarrow y$ in $SG(c)\}$
for $l := 1$ **to** "max. level of any node in $SG(c)$" **do**
"extend $<_s$ to totally order all level l nodes"
"further extend $<_s$ by setting $x <_s y$ for any x and y that
are descendents of level l nodes x' and y' respectively with $x' <_s y'$ "
end for

[$<_s$ is acyclic] To see this we use induction on l. The basis follows from the assumption that $SG(c)$ is acyclic. Suppose, by way of contradiction, that before iteration l $<_s$ is acyclic but afterwards it has a cycle among the nodes in the set C'. Let us instantiate $<_s$ and C' as the graph G and the set of vertices C in Lemma 3.3. The hypothetical cycle created by HTS at iteration l must include one of the new newly introduced edges and all new edges are between nodes whose least common ancestor is at level $l-1$ or closer to the root. Hence node $lca(C')$[2] must be at level $l-1$ or closer to the root. Now consider nodes p and p' that are children of the node $lca(C')$. The way HTS constructs $<_s$ guarantees that $p \rightarrow p'$ iff $v \rightarrow v'$ where v and v' are descendents of p and p' respectively. Hence the subgraph of $<_s$ whose vertices are the children of node $lca(C')$ corresponds to the graph G' in Lemma 3.3. By Lemma 3.3 a cycle among any set of nodes C' implies that there is a cycle among children of $lca(C')$, i.e., a cycle among nodes at level l or above, but HTS totally orders level l and it does not add edges above level l, a contradiction. Hence at iteration l no cycle can be created and $<_s$ produced by HTS is acyclic.
[Definition of c_s] We now construct a hierarchical tuple $c_s = (s_1, A, <_s^+, r, s_2)$ where $<_s^+$ is the irreflexive transitive closure of $<_s$.
[c_s is a CF] To show that c_s is a CF we have to prove that it satisfies (C6), (C7) and (C8). $<_s^+$ in c_s satisfies (C6) by construction: The iterated pseudo-command "further extend $<_s$ by setting $x <_s y$ for any x and y that are descendents of level l nodes x' and y' respectively with $x' <_s y'$ " of HTS guarantees this. Obviously extending $<_s$ to $<_s^+$ does not violate (C6). c_s satisfies (C7) trivially, since all transaction orders are captured by edges of type (b) of $SG(c)$ and embedded in $<_s$ by the first pseudo-command of HTS. By assumption c is a CF. Thus the DM projection of c, $c[D]$, is a D-computation. We

[2] The lowest common ancestor of a set of nodes C is the ancestor of every $x \in C$, but no proper descendent of x has this property. Assuming that the nodes of a CF are organized as a tree guarantees a uniform treatment since for any set of nodes there exist a lowest common ancestor.

now show that if x conflicts with and precedes y in $<$ then x precedes y in $<_s$. Recall that type (a) edges of $SG(c)$, which were originally installed between siblings that have conflicting descendents, are embedded in $<_s$, and HTS extended all these orderings to the descendents. Hence $<_s^+ [leaves(A)]$ is conflict consistent with $< [leaves(A)]$, and hence the DM tuple $c_s[D]$ is conflict equivalent to the D-computation $c[D]$, and by Lemma 3.2 $c_s[D]$ is a D-computation. Hence c_s satisfies (C8).

[c_s is serial] c_s is serial by construction, the iterated pseudo-command "extend $<_s$ to totally order all level l nodes" guarantees this.

[c is conflict serializable] By definition of c_s the two CFs have the same A, and as discussed above $c_s[D] \equiv c[D]$ hence c_s is conflict equivalent to c.

(only if) Suppose CF $c = (s_1, A, <, r, s_2)$ is conflict serializable, and hence there exists a serial CF c_s such that $c \equiv c_s$. For the sake of contradiction assume that $SG(c)$ has a cycle among siblings $x_1 \rightarrow x_2 \rightarrow \ldots \rightarrow x_k \rightarrow x_1$. Consider $x_i \rightarrow x_{i+1}$ in $SG(c)$. If $x_i \rightarrow x_{i+1}$ is a type (a) edge then since $c \equiv c_s$, c and c_s have conflict consistent order relations and therefore x_i appears before x_{i+1} in c_s. If $x_i \rightarrow x_{i+1}$ is a type (b) edge then since both executions have the same nodes and obey (C4), the transaction validity axiom, again x_i appears before x_{i+1} in c_s. Hence a cycle implies that each of $x_1, x_2, \ldots x_k$ appears before itself in the serial CF c_s, an absurdity. Thus no cycle can exist in $SG(c)$. That is, if CF c is conflict serializable then $SG(c)$ is acyclic. □

The graph $SG(c)$ that we have defined differs in two main ways from that defined by Hadzilacos and Hadzilacos[HH91] (which we refer to as $SG_{HH}(c)$). Firstly, SG_{HH} includes edges between incomparable nodes even if those nodes are not siblings. For example, condition (a) gives edges in SG_{HH} between any incomparable ancestors of x and y. Similarly condition (b) gives edges between any descendent of x and any descendent of y. Secondly, the graph SG_{HH} does not include vertices corresponding to the atomic operations of an execution; instead it only has vertices for subtransactions. The extra vertices for atomic operations seem essential for the definition to capture the intuitive notion of correct execution.

Figure 3.1 shows an instance of how SG_{HH} differs from our serialization graph. The upper part of the figure consists of three executions, c, c_{s_1}, and c_{s_2}. A filled circle denotes a transaction or an operation and a solid line denotes a caller-callee relationship. Nodes T_1 and T_2 are transactions. Nodes $w_1[x]$, $r_2[x]$, and $w_2[x]$ are atomic operations using the same notation as in [BHG87], with the same conflict relation, i.e.

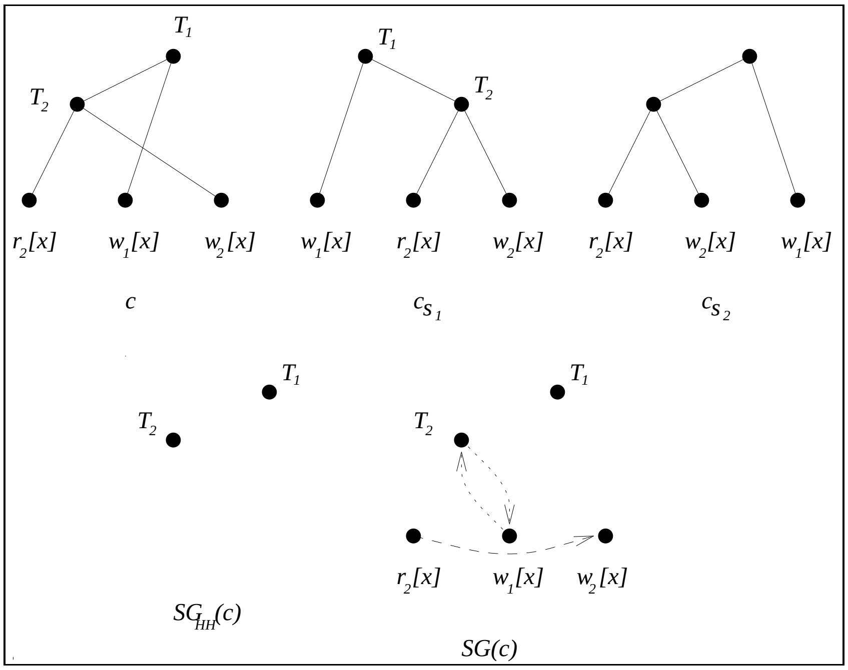

Figure 3.1: The importance of leaves as nodes.

two operations conflict if both are on the same object and at least one of them is a write (w) operation. It is easy to prove that c_{s_1} and c_{s_2} are the only serial executions involving the nodes of c. Also it is easy to show that c is not equivalent, as defined by [HH91], to c_{s_1}, or c_{s_2}, and hence c is not serializable. The lower part of the figure shows the serialization graphs $SG_{HH}(c)$ and $SG(c)$ of execution c. Execution c is non-serializable, however $SG_{HH}(c)$ is acyclic. Our serialization graph includes the operations and a cycle between T_2 and $w_1[x]$ and hence we reject the execution.

We now argue that if the serialization graph of [HH91] is fixed with respect to the problem mentioned then their graph accepts the same class of executions as ours. Once fixed, their serialization graph includes the same nodes as ours. Since they include more edges, someone could argue that it is possible that their graph can have a cycle among a set C' of nodes involving edges not defined in our graph. We will use Lemma 3.3 to show that cycle among a set C' of nodes in SG_{HH} will imply a cycle among children of $lca(C')$ in both SG and SG_{HH}. Hence if SG_{HH} has a cycle then SG also has a cycle. Instantiate SG_{HH} and C' as the graph G and set of vertices C in Lemma 3.3. The edges in SG_{HH} are such that $p \rightarrow p'$ iff $v \rightarrow v'$ where v and v' are are descendents of p and p' respectively. Hence the subgraph of SG_{HH} whose vertices are the children of node

$lca(C')$ corresponds to graph G' in Lemma 3.3. SG has an edge $v \to v'$ whenever v and v' are siblings and there exists an edge $p \to p'$ in SG_{HH} where p and p' are descendents of v and v' respectively. Hence the subgraph of our graph whose vertices are the children of node $lca(C')$ is identical to the same subgraph of SG_{HH}. By Lemma 3.3 a cycle among any set of nodes C' implies that there is a cycle among children of $lca(C')$, i.e. a cycle among nodes of a subgraph that is identical in both serialization graphs, SG and SG_{HH}. Hence any cycle in the serialization graph of [HH91] involving edges not defined in our serialization graph implies a cycle in sibling nodes involving the edges defined in SG. This means that the serialization graph of [HH91] naot only has redundant edges, but possibly has redundant cycles when rejecting executions.

The serialization graph of [FLW90] is defined in terms of the union of serialization graphs of individual transactions. A serialization graph for an individual transaction has its children as nodes and the edges are conflict and algorithmic edges. Hence their serialization graph does not have redundant edges nor redundant cycles, and it is quite similar to ours. The differences come only from the background models. In [FLW90] the serial correctness is defined as an external component, and not as the rearrangement of the same elements of an execution. Since their theory does not look exclusively in conflict serializability (they use serial correctness) it would not be consistent with their model to look for a result like ours.

3.4 An application of serializability theorem

We have extended from flat transactions the definitions and graph-based tool that make possible to prove conflict serializability in the context of nested transaction . There are different approaches to concurrency control protocols in the case of flat transactions. An interesting question to consider is whether flat transaction protocols have their counterpart in terms of nested transaction protocols. Locking is the most used technique for concurrency control protocols. The other two techniques are timestamps and graph testing. Such techniques are also employed to derive protocols that ensure conflict serializability of nested transactions. The work of [Ree78] describes a protocol for nested transactions that uses timestamp technique. The work of [Mos81] describes a protocol for nested transactions that uses lock techniques. Hadzilacos and Hadzilacos [HH91] show proofs of correctness for these two protocols. The work of [Jag90] describes a protocol that uses graph testing technique. We have proved [RA92] the correctness of this protocol in the framework of [HH91] and now we prove its correctness in our framework.

We will now describe the Children's Serialization Graph Testing, (CSGT) protocol [Jag90]. One interesting aspect of the protocol is the modular way it is defined. This protocol is the graph testing version of a serialization graph very similar to the serialization graph of [FLW90]. The proof of correctness we develop here indirectly connects our work to the work of [FLW90].

The *Children's Serialization Graph* of transaction t, $CSG(t)$, is a directed graph where nodes are the children of t. $CSG(t)$ has an edge $x \rightarrow y$ iff (a) there exists descendents x' and y' of x and y respectively, so that x' precedes and conflicts with y', or (b) $x <_t y$. It follows from the definition of $SG(c)$ and $CSG(t)$ that $CSG(t)$ corresponds to $SG(c)$ restricted to the nodes $children(t)$. Conversely $SG(c)$ corresponds to the union of the $CSG(t)$'s.

The CSGT rules are as follows. A graph, called the Stored Children's Serialization Graph of t, $SCSG(t)$, is created when a subtransaction t is invoked. Initially $SCSG(t)$ has no vertices. Each time t invokes a child y, t informs the scheduler about any other terminated child x such that $x <_t y$. This information gives the scheduler the ability to install type (b) edges. Each time a child y of t terminates, the scheduler adds the vertex y to $SCSG(t)$, adds the type (b) edges, and adds an edge $y' \rightarrow y$ ($y \rightarrow y'$) for each terminated sibling y' such that y' (y) or one of its descendents precedes and conflicts with y (y') or one of its descendents. Immediately after doing this, the scheduler tests if $SCSG(t)$ has cycles. If there are no cycles this means that up to now all children can be serialized. If a cycle is detected then CSGT signals y's abortion to t. It follows from the definition of $CSG(t)$ and $SCSG(t)$ that $CSG(t)$ is the same as $SCSG(t)$ when t terminates.

Theorem 3.2 *Every history produced by CSGT is conflict serializable.*

Proof. Let c be a CF representing an execution that follows the rules of CSGT. We will prove that $SG(c)$ is acyclic. For the sake of contradiction suppose that $SG(c)$ has a cycle, we will now prove that a cycle among any group C' of nodes implies a cycle in $SCSG(lca(C'))$. Let G, C, and P of Lemma 3.3 be instantiated as $SG(c)$, C' and the equivalence classes defined by the relation "descendents of the same child of $lca(C')$"

respectively. Observe that graph G' of Lemma 3.3 is a subgraph of $CSG(lca(C'))$. By Lemma 3.3 a cycle among nodes in C' implies that there is at least one cycle in $CSG(lca(C'))$. As observed above, when $lca(C')$ terminates $CSG(lca(C'))$ is the same as $SCSG(lca(C'))$, but CSGT checks for cycles for every child of a transaction, a contradiction. Hence $SG(c)$ cannot have cycles and by the serializability theorem c must be serializable. □

CSGT protocol can be easily modified to guarantee that the produced executions preserve the original ordering of nodes. More formally, we say that a CF c is *order-preserving serializable* if there is an equivalent serial CF c_s such that for each pair of nodes x and y if x is ordered before y in c then x is ordered before y in c_s. Lets call OP-CSGT the order-preserving version of CSGT. OP-CSGT is the same as the above described CSGT except for edges of type (b). Recall that in CSGT there exist a type (b) edge $x \rightarrow y$ iff x is algorithmically ordered before y. In OP-CSGT there exist a "type (b)" edge $x \rightarrow y$ iff x terminates before y. Since the stored graphs produced by OP-CSGT have the same nodes as the ones produced by CSGT and also at least the same edges, then OP-CSGT produces only serializable histories. OP-CSGT produces order preserving serializable executions because it ensures that the ordering of non-concurrent nodes are preserved.

Chapter 4

Semantic Serializability

4.1 Introduction

In Chapter 1 it was pointed out that one of the motivations for the use of nested transactions relates to object oriented databases. Object oriented databases differ from traditional databases by adopting some of the characteristics of object oriented systems [Kim90]. An object oriented database contains a set of objects. An object consists of variables and methods. These methods can invoke atomic operations on the object variables or invoke other methods. If we establish a correspondence between methods and transactions then the execution of methods maps conveniently into nested transactions [HH91].

In an object oriented database organized in the above way, we can identify the meaning of the method with its corresponding transaction. We will show that it is possible that at a lower level an operation conflicts with another, but because of the semantics of a method that uses directly or indirectly such operations, we can ignore the lower level conflict. The idea is similar to multilevel transaction systems [Wei91], but we will show that generic nesting imposes specific conditions. This chapter discusses the issues involved in using the meaning or semantics of a transaction in this sense. It is also possible to consider semantics in terms of groups of atomic operations [Kri93], or in terms of the context of an execution [Won93], but this is outside the scope of our work.

Object-oriented databases are being increasingly used to model non-standard applications. These applications place new constraints on databases design. In particular, they emphasize modularity, composition, and rapid prototyping. Often basic objects are defined, and more complex and advanced objects are composed recursively in terms of these basic objects. As a result, the traditional notion of conflict based on read and write operations may impose unnecessary constraints on the execution of transactions in such systems. To alleviate this problem, an appropriate approach is to use the semantics of higher level objects to ensure correct execution of transactions. Another aspect of an object-oriented database system is executions hierarchies that map nicely on nested transactions we have presented in Chapter 2. An operation invoked on a higher level object results in an entire execution hierarchy on lower level objects, sometimes referred to as composite objects [GK88]. This aspect has a nice mapping with the nested transactions we presented in Chapter 2. If traditional approaches such as strict two phase locking are used to manage composite objects, the execution hierarchies of two conflicting operations must be completely disjoint from each other. To remedy this problem, a protocol for executing transactions in object-oriented databases must allow execution hierarchies of different transactions to be interleaved as long as their behavior is equivalent to an execution in which these hierarchies are disjoint. Finally, conflicts among execution hierarchies of transactions are often not known *a priori* rather they are established dynamically during the execution. Protocols for transaction management in object-oriented databases need to address all these issues, viz., higher level semantics, nested executions, and dynamic conflicts.

In relational databases, the data is passive and has a flat structure. Nested transactions have been proposed in that context but the motivation is mainly for modularity of application and program design. On the other hand, in an object oriented databases, objects are active and have a nested structure. As a result, nested executions are of fundamental concern for transaction management in object oriented databases. The complexity of these nested executions is increased by the fact that in general one object can have its methods called by methods of more than one object, i.e., the involved objects are referentially shared [GK88].

Unlike relational databases where all interactions to the database is via read and write operations, objects have rich semantics associated with them. This semantics can potentially be used to increase concurrency among concurrent transactions. In this dissertation, we present locking protocols for transaction management in an object oriented databases. These protocols are novel since they incorporate the notion of semantics for executing nested transactions and are applicable for object oriented databases with referential sharing among disjoint objects. We believe this is an important property of object oriented systems since new objects are composed from existing objects, hence referential sharing among unrelated objects is unavoidable. Earlier work on transaction management in object oriented databases has either ignored the issue of semantics [HH91], or have only used semantics partially [AE92], or do not completely deal with referential sharing [MRW+93].

In this chapter, we present the model of object oriented database we have adopted, and we present the motivation for the development of such protocols.

4.2 Model

An *object oriented database* is a collection of *classes* and *instances* of these classes. A class defines a set of attributes for its instances and methods or procedures that are used to manipulate these instances. We assume that each class definition is physically represented as a *class object* and each instance is represented as an *instance object*. Thus, we use the generic term *object* to refer to both classes and instances of these classes. An object supports both atomic operations on its local state as well as methods to access both the local state as well as to access other objects. Users access objects by executing methods defined on objects or atomic operations associated with objects. A method execution is a partial order of method executions and atomic operations [HH91]. Such a caller-callee relationship between methods establishes a hierarchy that is known as *nested transactions* or *nested method executions* [Mos85]. These nested transactions are equivalent to the nested transactions presented in chapter 3 the only difference is that now we are specializing them to the object oriented database environment.

Figure 4.1 shows a database schema consisting of three classes of objects: Customer, Item, and Order. This example is similar to the one in [MRW+93] but is extended to a more realistic setting that supports referentially shared objects. Each customer can have several orders and each order may mention one or more items. Furthermore, each item could have been mentioned by several orders. Note that a particular object of class Order could be referentially shared among a customer and several items.

Figure 4.2 illustrates the skeleton of two methods. The class Customer defines a method "status-of-orders()" which invokes method "report-status()" on each of the

class Customer
customerNo: integer
name: string
orders: **set**(Order)

class Item
itemNo: integer
name: string
quantity: integer
orders: **set**(Order)
price: float

class Order
orderNo: integer
customer: Customer
status: integer
itemset: **set**(**tuple**(item: Item,
quantity: integer,
price: float))

Figure 4.1: Attributes of classes Customer, Item, and Order

```
method status-of-orders():
    ...
    for each orders objectID do
        ...
        objectID:report-status()
        ...
    od
    ...
end method;
```

```
method total-list-price():
    ...
    for each itemID in itemset do
        ...
        read(itemID.quantity )
        ...
        itemID.item:report-price()
        ...
    od
    ...
end method;
```

Figure 4.2: Examples of Methods

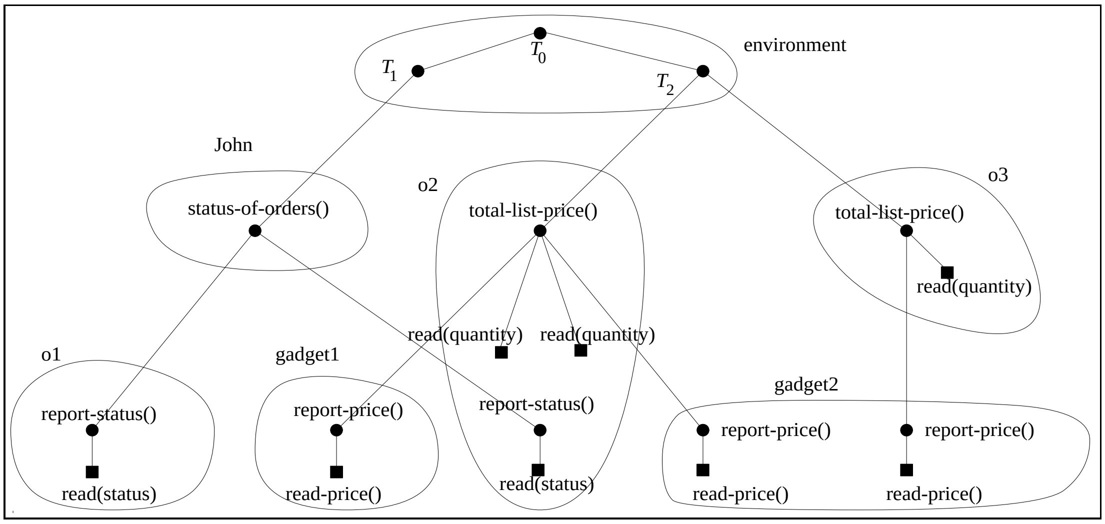

Figure 4.3: Concurrent execution of two transactions

orders of the customer where the status of an order indicates whether the order is pending, processed, etc. The class Order computes the total value of the order by simply accessing the quantity and price of each tuple in the set items. There is a provision for the price of some item to be different from the list price of that item, perhaps due to a discount. The method "total-list-price()" defined in the class Order computes the list-price for the order. In the class Order, the method "report-status()" is implemented with atomic operation "read(status)". Similarly in class Item, "report-price()" is implemented with atomic operation "read(price)" .

As in [HH91], we assume that user transactions are methods of a distinct object called *environment*. We will be using the terms *transaction* and *method execution* interchangeably. We also assume the existence of a special method execution, T_0, in the environment object that invokes the top-level method executions. Figure 4.3 shows an example of transactions causing nested calls of method executions in an object oriented database whose objects are instances of the classes in Figure 4.1. The execution order is from left to right. Method executions are represented by circles, and atomic operations are represented by squares. Transaction T_1 calls method "status-of-orders()" on Customer object John. The transaction T_1 is invoked with a key value such as a name customerNo and uses the select operation to retrieve the object-id of John. For the sake of simplicity, we omit such operations. John has two orders o1 and o2. Transaction T_2 computes the total list price of two orders o2 and o3. Order o2 is for two items gadget1 and gadget2 and o3 is for gadget2. Since there is one customer for each order, from the figure we can conclude that John is the customer of order o2 and has ordered gadget1 and gadget2.

Standard tree terminology is used to refer to relationships among method executions. For example method executions "total-list-price()" on objects o2 and o3 are *children* of T_2 in Figure 4.3. Node x is a *descendent* of y if $x = y$ or x is a child of a descendent of y. Node x is a proper descendent of y if it is a descendent other than y itself. *Ancestor* relationship is defined analogously.

In chapter 2 we have defined commutativity in terms of atomic operations. We now want to be able to derive a commutativity relation for internal nodes, i.e. for method executions. Two method invocations x and y on the same object commute if and only if the two possible sequential executions of x and y are indistinguishable for both x and y and for all possible sequences of methods that may be invoked subsequently [CF90, HW91, MRW+93]. Table 4.1 illustrates the commutativity relationships between the methods of class object *Item*. The entry $\perp$ indicates that the relationship is undefined. It is not necessary to derive commutativity relationships for all pair of methods of an object.

decrease-percent()	report-price()	change-price()	
commute	$\perp$	$\perp$	decrease-percent()
$\perp$	$\perp$	$\perp$	report-price()
$\perp$	$\perp$	$\perp$	change-price()

Table 4.1: Commutativity Table for Object Class *Item*

We assume that the commutativity relationships between the operations are well-defined and can be derived based on the semantics and the specifications of the class and its methods. However, in a dynamic object oriented database system, methods on different objects may invoke methods on the same objects resulting in conflicts (specially when referential sharing is allowed). Such a conflict cannot be determined a priori based on the static analysis rather it materializes as the particular methods are executed.

Beeri, Bernstein, and Goodman [BBG89] developed a general theory to reason about nested executions with semantics that is summarized in Chapter 2. They have an equivalence concept that we call *semantic equivalence* in contrast to our *conflict equivalence* presented in chapter 3. In their work they also introduced two transformations that establish semantic equivalence relationship between executions: *substitution* and *commutativity-based reversals*. *Substitution* can be specialized in terms of reduction and expansion. Using sequences of these two transformations we can establish the semantic equivalence of some execution and its serial counterpart, proving serializability of nested executions. A transaction x is *separated* if no other operations of other transactions are interleaved with its leaves. Consider a transaction x where all its children are leaves and it is separated. *Reduction* is a transformation that relates two nested executions where the first one has a separated transaction x whose children are leaves and the second is the same as the first except that x was substituted by the atomic operation corresponding to transaction x. The underlying assumption is the existence of a correspondence between a transaction and an atomic operation, i.e., if the execution of a transaction is atomic then there exists a corresponding atomic operation. *Expansion* is the inverse of reduction. The transformation *commutativity-based reversal* relates two computations where the first has two consecutive leaves that commute and the second is the same as the first except that these two consecutive leaves are reversed.

4.3 Locking

Locking is one of the most used mechanisms to solve the problem of synchronizing access to shared resources. We will use the framework of locks and locking protocols to present our approach relative to the use of semantic information in the synchronization of nested transactions. We will first present Moss' two-phase locking protocol. Then we show how semantic information can be used with the help of an example. We finish this chapter by presenting the work of Muth et al [MRW+93], that has similar objectives as part of our work, but has some problems.

Locks are place holders for checking concurrency control predicates. Each atomic operation has a lock associated with it. Before a method execution may execute an atomic operation, a lock must be requested and granted. We say that a method that executes an atomic operation x *holds* lock $lock(x)$. We say that $lock(x)$ *conflicts with* $lock(t)$, if atomic operation x conflicts with atomic operation t. When a method execution t terminates, its parent *inherits* all the locks t owns. A special case occurs when the terminating method execution is a top-level one. In this case the locks are released, i.e. they cease to exist. The locks that a method execution x inherit from method executions that are its children are said to be *retained* by x. This means that there are two ways for a method execution to own a lock. Method execution x' can own $lock(x)$ in the sense of holding, meaning that atomic operation x is executed directly by x'. Method execution x' can own $lock(x)$ in the sense of retaining, meaning that atomic operation x was executed by some proper descendent of x'.

Lock $lock(x)$ can be granted to method execution x', i.e. atomic operation x can be executed by x', iff no other method execution holds a conflicting lock, and all the retainers of a conflicting lock are ancestors of x'. We summarize the nested two-phase locking (N2PL) rules adapted from [Mos85] as follows.

1. A method execution t' can execute an atomic operation t, iff $lock(t)$ is requested and is granted (we say that the method execution t' *holds* $lock(t)$).

2. A method execution cannot terminate (commit or abort) until all its children have terminated. When a method execution terminates:

   (a) *It is not top-level and it commits:* Its locks are inherited by its parent (we say that the parent *retains* the inherited locks).

   (b) *It is not top-level and it aborts:* Its locks are discarded.

   (c) *It is top-level:* Its locks are discarded.

3. A lock $lock(x)$ can be granted to a method execution iff

   (a) No other method execution holds a conflicting lock, and

   (b) All the retainers of a conflicting lock are ancestors of the requesting method execution.

It is shown in [BBG89, HH91] that N2PL ensures serializability. Since N2PL does not take advantage of any knowledge about the semantics of the methods being executed, we can prove its correctness using the techniques of Chapter 3. We will now show an example of how the semantics of method executions can be used in order to ignore certain conflicts and accept more executions.

Suppose that the class Item of Figure 4.1 has a method "decrease-percent()". This method decreases the price of its corresponding item. Observe that two instances

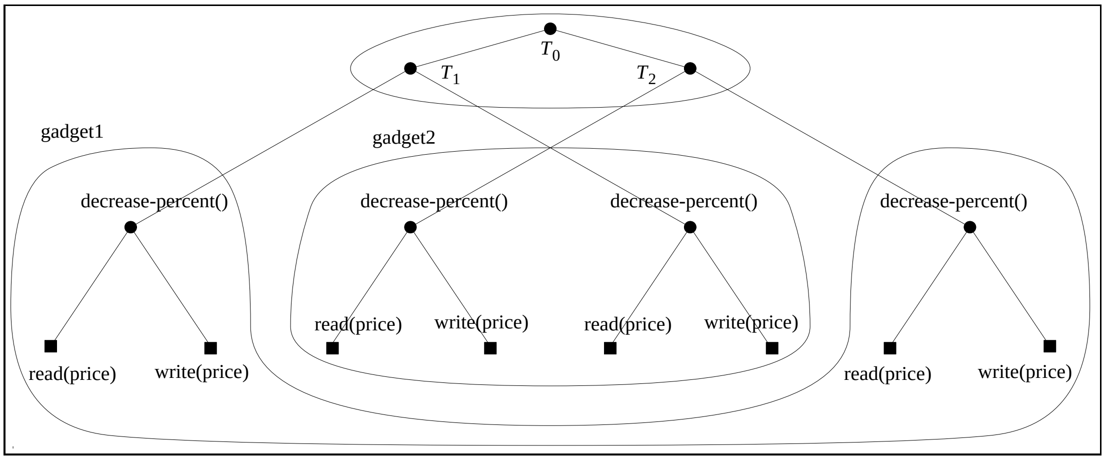

Figure 4.4: A semantically serializable execution

of method "decrease-percent()" commute since percentage operations commute. Figure 4.4 shows two transactions T_1 and T_2 that execute methods "decrease-percent()" concurrently on gadget1 and gadget2. Observe that method "decrease-percent()" is implemented using two atomic operations "read(price)" and "write(price)". In the traditional conflict serializability that does not take advantage of semantics of higher level operations in nested transactions, this execution would be considered non-serializable. In this execution, transaction T_1 must be serialized before transaction T_2 because of the conflicts in gadget1 and transaction T_2 must be serialized before transaction T_1 because of the conflicts in gadget2. However, if the semantics of "decrease-percent()" is taken into consideration (see Table 4.1), two such operations commute and hence their relative order is not significant. Therefore, this execution will be serializable and is equivalent to both T_1 before T_2 or vice-versa.

There are cases where semantic information cannot be used to ignore conflicts at the lower level. We start by showing a very simple occurrence that prohibits the use of semantic information. Consider another execution of the same two transactions that is illustrated in Figure 4.5. In this execution we cannot apply the commutativity information between the two "decrease-percent()" method executions on gadget2. This is because the commutativity of these two method executions is derived assuming the atomicity of the operations. In this execution the two method executions are not

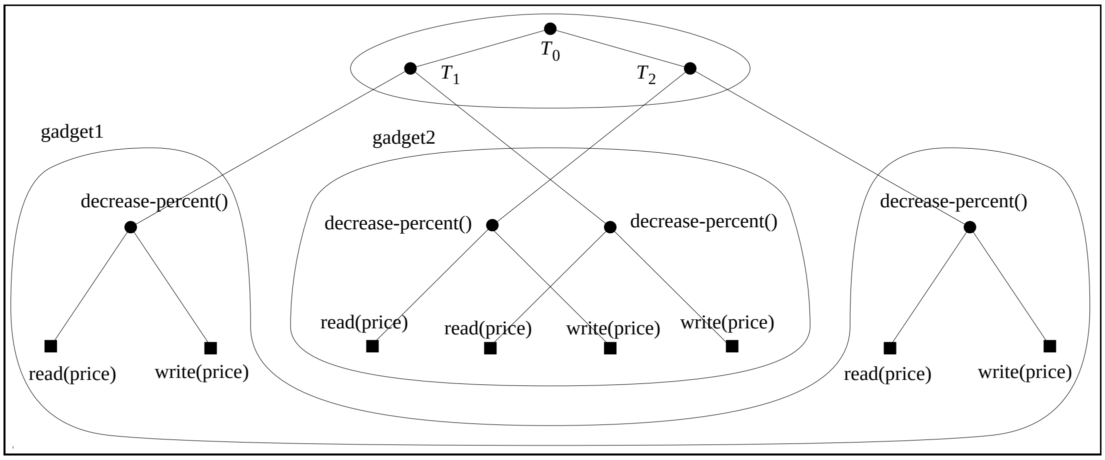

Figure 4.5: A nonserializable execution

executed atomically with respect to each other.

Using the theory described in [BBG89] we can establish the semantic serializability of the execution showed in Figure 4.4. We can *reduce* the computations rooted at method executions "decrease-percent()" since these computations are separated. Now we can use the semantics of the Item object to apply *commutativity-based reversals* on the reduced method executions "decrease-percent()" of gadget2. Basically nested two phase locking does not have any mechanism that allows lower level conflicts (such as "read(price)", "write(price)" in Figure 4.4) to be ignored by using the semantics at higher level methods (e.g., "decrease-percent()").

Muth et al. [MRW+93] extend N2PL to use semantic information in the following way. First they assume that they have, for each object, compatibility matrices with entries for all the methods. An entry in such a compatibility matrix informs if the corresponding methods conflict or commute. Their locking protocol prescribes locks not only for the atomic operations but also for the methods. They have rules for lock inheritance similar to the ones in N2PL. When a method terminates its parent inherits its locks. When a method is a top-level transaction all its locks are released. Basically, a lock of appropriate type must be obtained before executing a method. A lock is granted in three cases. First when there are no other conflicting locks. Second when there are conflicting locks but they are siblings of the requested lock. Third when there are conflicting locks but the semantic information about ancestors allow the granting of the lock. They specify that a requested lock r is not blocked by a conflicting lock h if there exists an ancestor h' of h that commutes with an ancestor r' of r, and h' has terminated. The first and second case is similar to N2PL, the third case corresponds to the use of semantics. They state that their protocol does not apply to referentially shared objects.

We now show that the interplay between semantic information and hierarchy of methods, can pose challenging problems for protocols that want to use semantic information. We will show an execution that is accepted by the protocol of Muth et al [MRW$^+$93] and nevertheless is not semantically serializable. It is a simple execution but nevertheless illustrates the type of problems that may occur.

In order to maintain the execution simple but also intuitive, we will abandon for a while our classes Item/Order/Customer and resort to a database constituted of two classes of objects, class **arith** and class **pdec**. Objects of class **arith** are composed of objects of class **pdec**. The methods of these classes are shown below and there are no referentially shared objects.

Class **pdec** implements persistent decimal numbers and the supported operations are "read", denoted by $r()$, and "write", denoted by $w()$. Table 4.2 shows the compatibility ralation for methods of class **pdec**.

pdec	$r()$	$w()$
$r()$	ok	conflict
$w()$	conflict	conflict

Table 4.2: Compatibility Table for Object Class **pdec**.

Class **arith** implements some updating operations using an object of class pdec. We will use as the names of the methods of class **arith** the standard mathematical expressions corresponding to the meaning of the arithmetical operation performed (we will use the generic name x for an instance). Class **arith** has six methods. Method x corresponds to the identity function, $x := x$. Method $x+1$ corresponds to an increment, and so on. Actually method x has two implementations: method x_1 and x_2. Obviously these two implementations are correct and have the same semantics, and consequently x_1 and x_2 have the same entries in the compatibility table. The methods and their

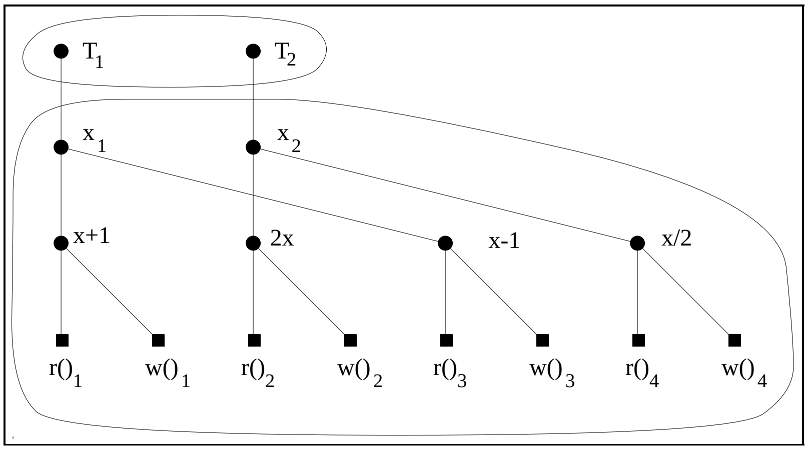

Figure 4.6: A nonserializable execution, bis

compatibility relationship for class **arith** are given by Table 4.3.

arith	x_1	x_2	$x+1$	$x-1$	$2x$	$x/2$
x_1	commute	commute	commute	commute	commute	commute
x_2	commute	commute	commute	commute	commute	commute
$x+1$	commute	commute	commute	commute	conflict	conflict
$x-1$	commute	commute	commute	commute	conflict	conflict
$2x$	commute	commute	conflict	conflict	commute	commute
$x/2$	commute	commute	conflict	conflict	commute	commute

Table 4.3: Compatibility table for object class **arith**

Consider the execution shown in Figure 4.6. We have added subscripts in order to identify multiple occurrences of the same operation or method, e.g. $r()_2$ refers to the second occurrence of operation $r()$ and is a child of $2x$, and the latter is a child of x_2. N2PL would not accept this execution. Since $lock(r()_2)$ conflicts with $lock(w()_1)$ the N2PL scheduler grants $lock(r()_2)$ only when T_1 terminates. One of the possible orders that the protocol of Muth et al. [MRW$^+$93] grants the locks in the execution of Figure 4.6 and its justification is as follows.

1. $lock(x_1)$, no other conflicting locks.

2. $lock(x+1)$, no other conflicting locks.

3. $lock(r()_1)$, no other conflicting locks.

4. $lock(w()_1)$ conflicts with $lock(r()_1)$ but they belong to the same transaction. Then $x+1$ completes.

5. $lock(x_2)$, no other conflicting locks.

6. $lock(2x)$ conflicts with $lock(x+1)$ but they have ancestors x_2 and $x+1$ (completed) that commute.

7. $lock(r()_2)$ conflicts with $lock(w()_1)$ but they have ancestors x_2 and $x+1$ (completed) that commute.

8. $lock(w()_2)$ conflicts with $lock(r()_2)$ but they belong to the same transaction. It conflicts with $lock(r()_1)$ and $lock(w()_1)$ but they have ancestors x_2 and $x+1$ (completed) that commute. Then $2x$ completes.

9. $lock(x-1)$ conflicts with $lock(2x)$ but they have ancestors x_1 and $2x$ (completed) that commute.

10. $lock(r()_3)$ conflicts with $lock(w()_1$ but they have ancestors $x-1$ and $x+1$ (completed) that commute. It conflicts with $lock(w()_2)$ but they have ancestors x_1 and $2x$ (completed) that commute.

11. $lock(w()_3)$ conflicts with $lock(r()_3)$ but they belong to the same transaction. It conflicts with $lock(r()_1)$ and $lock(w()_1)$ but they have ancestors $x-1$ and $x+1$ (completed) that commute. It conflicts with $lock(r()_2)$ and $lock(w()_2)$ but they have ancestors x_1 and $2x$ (completed) that commute. Then $x-1$, x_1, and T_1 complete.

12. $lock(x/2)$: (T_1 has terminated and its corresponding locks are released) no other conflicting locks.

13. $lock(r()_4)$ conflicts with $lock(w()_2)$ but they have ancestors $x/2$ and $2x$ that commute.

14. $lock(w()_4)$ conflicts with $lock(r()_4)$ but they belong to the same transaction. It conflicts with $lock(r()_2)$ and $lock(w()_2)$ but they have ancestors $x/2$ and $2x$ that commute. Then $x/2$, x_2 and T_2 complete.

As shown by the above sequence, the execution of Figure 4.6 is accepted by the protocol of Muth et al. [MRW$^+$93], but nevertheless it does not ensures the semantic serializability of the execution. The execution of Figure 4.6 is not semantically equivalent to any serial order T_1T_2 or T_2T_1 (both orders are equivalent since T_1 and T_2 invokes identity update. For example if the primitive object of class **pdec** start with value 0.5 at the end of the execution instead of continuing to have the value of 0.5 it gets 1.

The protocol of [MRW$^+$93] suffers from better formalization. The area of concurrency control is famous with respect to how treacherous can be a solution that seems "reasonable and intuitive". A proof of correctness of their protocol is not presented, and this is of fundamental importance. In the next chapter we introduce semantic locking protocols that ensures semantic serializability.

Chapter 5

Semantic Locking Protocols

In traditional databases, the most commonly adopted correctness criterion is serializability. Moss [Mos85] introduced the notion of nested transactions for modular composition of applications and programs in database systems. He proposed the nested two phase locking protocol (N2PL) which is a generalization of the well-known two-phase locking protocol for concurrency control [EGLT76].

We have developed a locking-based concurrency control protocol for object-oriented databases. Our protocol generalizes Moss's original nested transaction protocol in that it incorporates the rich semantic information available in object oriented databases. Our protocol incorporates the notions of semantics of higher level objects [BBG89, MRW+93], nested executions [HH91, AE92, MRW+93] and dynamic conflicts. In particular, we address the issue of referentially shared objects where the referential sharing among objects is determined dynamically during the execution. The correctness of our protocol is based on the theory of nested transactions proposed in [BBG89].

5.1 Using semantics of methods

We start by describing some of the salient aspects of object-oriented databases that are considered in the design of our protocol. In general, in object-oriented databases, the conflicts and commutativity relations between methods are defined on a per class basis. Methods on different objects are generally assumed to commute with each other [GK88, CF90, HH91, MRW+93]. This approach is reasonable for object-oriented databases with disjoint complex objects. However, in a system with referentially shared sub-objects, the commutativity relation needs to be redefined even for methods that are executed on different objects. One approach could be to define these relations across objects statically but this approach could be very conservative and defeats the goal of modularity and extensibility in object-oriented databases. Furthermore, the referential sharing of objects may occur dynamically during execution.

Our assumptions are as follows. We assume that for each class, conflict relations are defined for the atomic operations. This is the same assumption made by most previous work in both nested transactions and object oriented databases [Mos85, HH91, FLW90]. However, we enrich this model by exploiting semantic information, i.e., commutativity relations that are derived from the semantics of methods of classes. Specifically we assume that commutativity information about pairs of methods is potentially available. Such methods are known to commute a priori. The conflict and commutativity relations between methods that referentially share sub-objects and methods that do not commute a priori are left unspecified and are determined dynamically during the execution of such methods. Thus, tests for conflicts are done only for methods executing at the same object. Methods unless algorithmically ordered execute concurrently and depending on their executions, the proposed protocol will capture the conflict relations among such methods. An important feature of such approach is that it does not require the designer to derive commutativity relations for all methods of all classes. The designer provides semantic information at his/her own discretion.

5.2 Semantic locking protocol

5.2.1 Protocol

In semantic locking each atomic operation has a *lock* associated with it. Before a method execution may execute an atomic operation, a lock must be requested and granted. This is similar to [Mos85], locks are only required for the execution of atomic operations and contrasts with [MRW+93] where locks are required for the execution of methods. Similar to [Mos85] when a method execution terminates its parent inherits all its locks. The locks that a method execution x inherits from its children are said to be *retained* by x. In semantic locking, locks can be granted to conflicting operations if the corresponding operations have commuting ancestors.

The important observation to make regarding semantic information is that two methods, which are known to commute, commute only if both are executed atomically. Hence in Figure 4.4 a pair of "decrease-percent()" commutes if the operations implementing these methods do not interleave. Once all the operations implementing one "decrease-percent()" are executed the operations of another "decrease-percent()" can execute. We implement this idea by allowing "decrease-percent()" of T_1 to acquire the lock for "read(price)" only after the locks on "read(price)" and "write(price)" are inherited by T_2, i.e., "decrease-percent()" has terminated. In particular, a lock request $lock(t)$ which conflicts with another lock, $lock(x)$ can be granted using semantic information only when $lock(x)$ is inherited by a node whose descendent commutes with an ancestor of $lock(t)$. In this case, we say that the pair of commuting ancestors *relieves* the conflict among atomic operations. A more formal definition will be given ahead.

Another important point is that when semantic information is used we have to be careful about hierarchical constraints. As an instance, when a conflict between descendents of methods x and y is ignored because method x commutes with method y the conditions that enabled the generation of the information about commutativity must be guaranteed. Recall that we abstract from implementations issues and define that method x commutes with method y if and only if the two possible sequential executions of x and y are indistinguishable for both x and y and for all possible sequences of methods that may be invoked subsequently. Hence we have to guarantee that when semantic information is used the implied atomicity is fulfilled. For instance if x is related to y we cannot guarantee their atomicity and hence to relieve a conflict a pair of nodes needs not only to commute but also to satisfy certain hierarchical restrictions. We have to enforce these restrictions not only in a pairwise manner but transitively too.

The non-atomic behavior of non-related executions such as the one in Figure 4.5 is captured at the level of the two phase lock rule. Non-atomic behavior of computations rooted at related nodes or transitively related because of conflict relieving is not captured by the two phase lock rule. Thus we have to add rules that cope with the issue of atomicity of related commuting nodes. To formally state the additional rules we introduce some definitions.

Node a is a *directly commuting ancestor* of node b iff a and b commute and a is an ancestor of b.

Node a is an *indirectly commuting ancestor* of b if there is a sequence of nodes $a_1, a_2, ...a_{2n}$, $n \geq 1$, such that

- a is a proper ancestor of a_1, and a_1 and a_2 commute; and
- for $1 < k < n$, a_{2k} is a proper ancestor of a_{2k+1}, and a_{2k+1} and a_{2k+2} commute; and
- a_{2n} is a proper ancestor of b.

We now describe the protocol which is referred to as *semantic locking*. We introduce the following notation for intervals of related nodes to simplify the presentation of the protocol. Let x be a method execution and let x' be a descendent of x, $[x \cdots x']$ represents the set $\{a \mid a \in descendents(x) \wedge a \in ancestors(x')\}$. If we substitute '[' or ']' by '(' or ')' the proper descendents/ancestors should be taken e.g. $(x \cdots x'] = [x \cdots x'] - \{x\}$. A method execution must observe the following semantic locking rules:

1. A method execution t' can execute an atomic operation t, iff $lock(t)$ is requested and is granted (we say that the method execution t' *holds* $lock(t)$).
2. A method execution cannot terminate (commit or abort) until all its children have terminated. When a method execution terminates:
   (a) *It is not top-level and it commits:* Its locks are inherited by its parent (we say that the parent *retains* the inherited locks).
   (b) *It is not top-level and it aborts:* Its locks are discarded.
   (c) *It is top-level:* Its locks are discarded.
3. A lock $lock(t)$ can be granted to a method execution iff
   (a) No other method execution holds a conflicting lock, and
   (b) All retainers, x'', of a conflicting $lock(x)$ are such that
      - x'' is an ancestor of t, or
      - x'' is not an ancestor of t, and some element x' of $(x'' \cdots x]$ and some ancestor t' of t commute, where t' is not a directly or indirectly commuting ancestor of x'.

According to Rule 1, locks must be acquired only for the leaves (atomic operations). Rule 2 enforces the strict two phase locking restriction for nested executions. Rule 3 (a) is the two phase locking rule, and rule 3 (b) relaxes the nested two phase locking constraint by using a possibly available commutativity relation between method executions based on semantic information. In nested two phase locking, a lock can be acquired provided:

- No other method execution holds or retains a conflicting lock; or
- if there are conflicting locks, such locks are retained by the ancestors.

Semantic locking grants a lock in the above cases as well as in the following case:

- If there are conflicting locks retained by non-ancestors then one of the ancestors of the operation up to but not including its retainer and some ancestor of the requester commute and these commuting nodes are not directly or indirectly related.

In this latter case semantic information is used by the locking protocol, hence the name semantic locking protocol.

5.2.2 Analysis

We underscore the distinction between nested two phase locking and semantic locking with the help of an example. In particular, recall the execution illustrated in Figure 4.4. Nested two phase locking cannot derive this execution. In particular, T_1 is blocked by T_2 at the level of gadget2. Method execution "decrease-percent()" of transaction T_1 is blocked trying to acquire a lock on method "read(price)" (it conflicts with the lock on method "write(price)" held by "decrease-percent()" of transaction T_2). Similarly, T_2 is blocked by T_1 at the level of gadget1. Method execution "decrease-percent()" of transaction T_2 is blocked trying to acquire a lock on method "read(price)" and hence deadlocked. When semantic locking is used for this execution such deadlocks will not occur. Consider for example the method executions of T_1 and T_2 on the Item gadget2. When T_1 executes its "read(price)" on gadget2, it conflicts with the "write(price)" executed by T_2. However, the semantic locking rule permits "decrease-percent()" of T_1 to acquire a lock for executing "read(price)" on gadget2 after "decrease-percent()" of T_2 terminates and the lock corresponding "write(price)" on gadget2 is retained by T_2. Since commutativity of "decrease-percent()" is defined for completed operations, the "decrease-percent()" executed by T_1 must wait until the "decrease-percent()" executed by T_2 has completed for the commutativity property to hold. This explains the reason for excluding the retainer x in the expression $(x \cdots x']$ of Rule 3 in the protocol. However, note that even with semantic locking, deadlocks are possible. For example, in Figure 4.5, semantic locking results in a deadlock. This occurs since both T_1 and T_2 are both blocked when trying to obtain locks on gadget2.

To reason about the correctness of semantic locking we use the two transformations described in [BBG89]. We illustrate the use of these two transformations in Figure 5.1 using the example of Figure 4.4. Since each "decrease-percent()" in execution (a) is separated we can reduce them. We can go from execution (a) to execution (b) by using four reductions. Even though we maintain the labels "decrease-percent()", in execution (b) these nodes are actually substituted by the corresponding atomic operations. In execution (b) we can apply commutativity-based reversal in order to separate T_1, coincidentally we also separate T_2 and we obtain execution (c). In execution (c), since T_1 and T_2 are separated we can use two reductions to obtain execution (d) which is a serial execution.

We now use the bottom-up proof paradigm of [BBG89] to informally argue about the correctness of semantic locking. The goal is to create a sequence of equivalent executions starting from an execution, c_0, resulting from semantic locking and terminating with a serial execution, c_{serial}, consisting of top-level transactions. The proof paradigm is to establish a sequence of equivalent executions and the intermediate executions in this sequence are referred to as c_i. We choose a node, separate it using as many commutativity-based reversals as necessary, and then reduce it. We repeat this separation and reduction process until only the top-level method executions are left. With this we establish the equivalence of the execution produced by semantic locking with a serial execution consisting of only the operations corresponding to the top-level method executions.

In each iteration we choose for the separation and reduction the node x whose rightmost child is the leftmost among nodes that have only leaves as children. We have to prove that it is always possible to separate x. We say that an operation *interleaves* within x if there are children of x at the left as well as the right of this operation. Our proof obligation is to show that in spite of such interleavings, there always exists such an x that can be separated. For c_0 no operation t that conflicts with a child of x

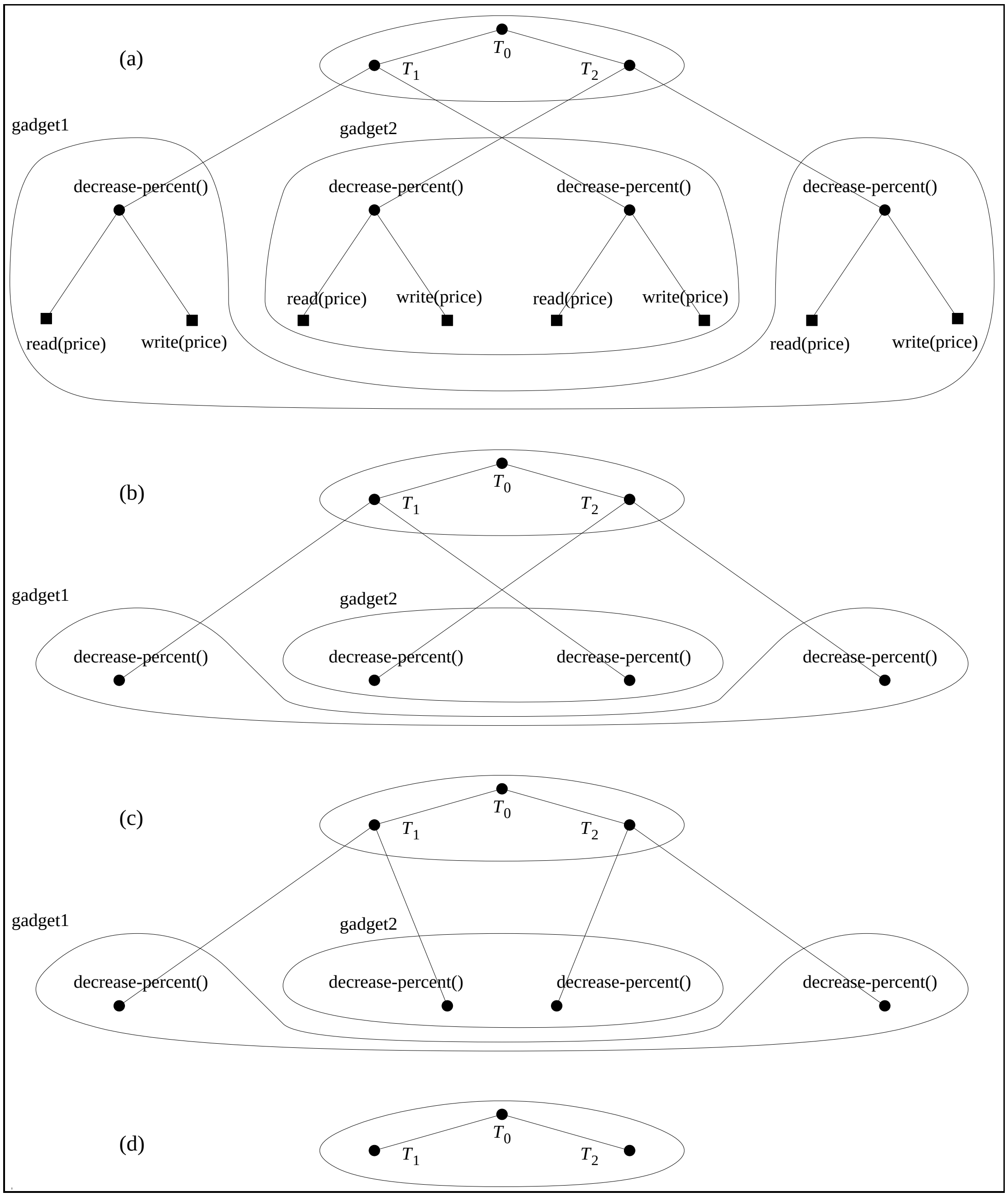

Figure 5.1: Using commutativity-based reversals and reduction

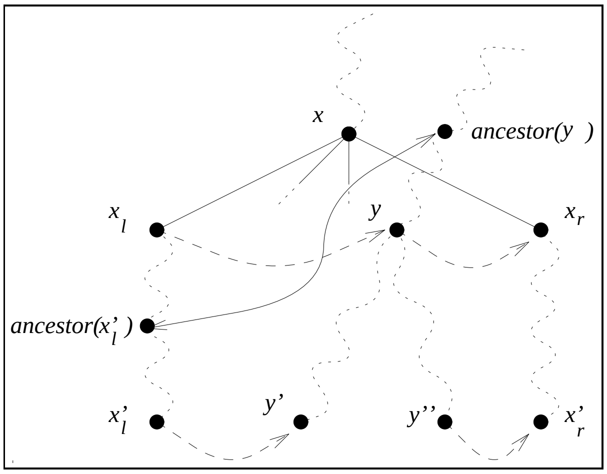

Figure 5.2: A hypothetical node y blocks x's separation.

interleaves within x. This is the case since from rule 3 of the protocol t must wait at least until x has terminated. We now need to argue that this separation and reduction can be performed when we are dealing with any intermediate execution c_i.

For the sake of contradiction assume that in c_i we cannot separate x whose rightmost child is the leftmost among nodes that have only leaves as children. This may be because x_l, a child of x, conflicts with and precedes y and that y conflicts with and precedes x_r, another child of x as shown in Figure 5.2. Without loss of generality assume that this conflict is caused in c_0 by x'_l and y' as shown in the figure. Assume that this conflict was not relieved. Then y' must have waited until the lock on x'_l was released or inherited by an ancestor of y'. In both cases, and due to the choice of x, all descendants of x must have terminated before y' was executed. Hence, y could not have been interleaved within the children of x. Next assume the conflict was relieved by a pair: ancestor of x'_l, $ancestor(x'_l)$, and a proper ancestor of y, $ancestor(y)$, as shown in the figure. Node y conflicts with and precedes x_r, without loss of generality assume that the conflict was caused by y'' and x'_r as shown in Figure 5.2. The conflict of y'' and x'_r is either relieved or not. If we assume that the conflict is relieved then an ancestor of y'' that is a descendent of $ancestor(y)$ and an ancestor of x is the pair relieving the conflict. But components of such a pair are indirectly commuting ancestors, hence this cannot happen. The other alternative is that the conflict is not relieved. Because of the way the nodes are separated and reduced this would imply that y is an ancestor of x. Since y is an ancestor of x then $ancestor(y)$ is a directly commuting ancestor of $ancestor(x'_l)$ and a new violation is derived. Thus such execution can not exist. This argument can be extended to the general case when a sequence of nodes $y_1, \ldots, y_n$ are interleaved within x. A formal and complete proof is shown in section 5.2.3.

If semantic locking is presented with the execution of Figure 4.6 the protocol would behave in the following way. The locks for nodes $r()_1$ and $w()_1$, are granted exactly like in N2PL. Lock $lock(r()_2)$ conflicts with $lock(w()_1)$, and in N2PL it would be granted only after T_1 terminates. Semantic locking grants $lock(r()_2)$ to method execution $2x$ because $x+1$ (already terminated) commutes with x_2. Lock $lock(w()_2)$ is also granted based on the fact that $x+1$ and x_2 commute. Lock $lock(r()_3)$ conflicts with $lock(w()_1)$ but this is not a problem since the latter is retained by an ancestor of $lock(r()_3)$. Lock $lock(r()_3)$ conflicts with $lock(w()_2)$ and even though they have commuting ancestors, x_1 is an indirectly commuting ancestor of $2x$ (and x_2 cannot be used because it is not terminated yet). Hence the granting of $lock(r()_3)$ will be blocked until at least x_2 terminates avoiding therefore this non-serializable execution.

5.2.3 Proof of Correctness

In this section we prove a theorem that shows that semantic locking ensures correctness, i.e., semantic locking only accepts semantic serializable execution.

Theorem 5.1 *Semantic locking allows only serializable executions.*

Proof. We first give a summary of the proof. We will use the theory in [BBG89] to show that all executions produced by semantic locking are equivalent to a serial execution. Let c_1 be an execution produced by semantic locking. We will show that, starting with c_1, it is possible to generate a sequence of equivalent executions, where the last one, c_n, is a serial execution. More precisely, c_n consists of a two level tree where the root is T_0 and its children are the nodes corresponding to the top-level method executions of c_1.

Consider the sequence of executions produced by the execution of the following algorithm. We will refer to it as *Separate Leftmost and Reduce* algorithm, or SLR algorithm:

$i \leftarrow 1$

$c_i \leftarrow$ "Execution originally produced by Semantic Locking"
while "c_i does not consist of T_0 and its children" **do**

  $x_i \leftarrow$ "The node of c_i whose rightmost child is the leftmost among the nodes that have only leaves as children"

  "Separate and reduce x_i"

  $i \leftarrow i + 1$

end while

Node x conflicts with node y in an execution c_i if and only if, in execution c_1, x has descendent x' that conflicts with y' a descendent of y, and no ancestors of x' that are descendents of x commute with ancestors of y' that are descendents of y. We say that x and y are *conflicting nodes.*

We say that nodes X are *interleaved within* x in an execution c_i when x is a node that has only leaves as children, nodes in X are leaves, and there exist children of x that precede and follow each node in Y. We claim that it is always possible to separate x_i using commutativity-based reversals. Consider the nodes that are interleaved within x_i. The commutative-based reversals will be done such that two groups of nodes will maintain their relative order. Each node interleaved within x_i will belong to one of two groups: $LeftGroup$ or $RightGroup$.

We can classify nodes interleaved within x_i in 4 sets.

1. The $direct - conflict$ set,
2. The $indirect - conflict$ set,
3. The $indirect - algorithm$ set,
4. The $inert$ set.

We can especialize the sets, with the exception of $inert$ in terms of how they interact with children of x_i. If they conflict with and precede a child of x directly or indirectly then it belongs to a left set, otherwise it belongs to a right-set. This criteria creates 7 different categories of nodes. We will show that nodes interleaved within x_i belongs either to the left category or to the right but not to both.

A node y interleaved within x_i belongs to the $LeftGroup$ iff

- y conflicts with and precedes a child of x_i.

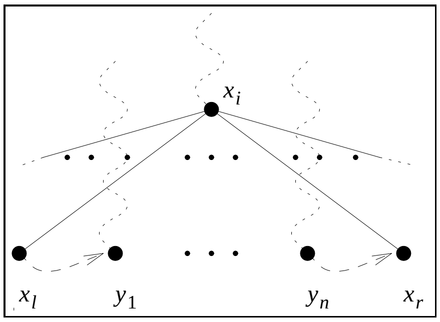

Figure 5.3: A hypothetical non-separable x_i.

- y conflicts with and precedes node $y' \in LeftGroup$
- y algorithmically precedes $y' \in LeftGroup$

A node y that is not in $LeftGroup$ is in *RightGroup*. Hence a node in *inert* will be moved to the right.

In what follows we will be involved in reasoning about a given execution after it was transformed. We will have to reason about a given execution that is not the one produced by semantic locking. We will reason about an execution that is produced after some commutative reversals (for separation) and reductions using the SLR algorithm are applied to an execution produced by semantic locking.

Lemma 5.1 *For every c_i, node x_i can be separated and reduced as prescribed above.*

Proof. The only reason why x_i in an execution c_i cannot become separated is when there exists a sequence of leaves $y_1, \ldots y_n$ related among them and to two children, x_l and x_r of x_i in the following way:

1. x_l conflicts with and precedes y_1,
2. for $1 \leq j < n$, y_j conflicts with and precedes y_{j+1} or y_j algorithmically precedes y_{j+1}, and
3. y_n conflicts with and precedes x_r.

We refer to $y_1, \ldots, y_n$ or to a subset of them as "y's".

Figure 5.3 shows nodes $y_1, \ldots, y_n$ interleaved within x_i. The labeled filled circles indicate method executions or atomic operations, e.g. the upper filled circle represents method execution x_i. The solid lines indicate direct caller-callee relationship, for example since x_i directly calls x_l they are connected by a solid line. The ellipsis represented by small filled circles indicates either the other y's interleaved within x_i, or the other possible children (and implicit caller-callee relationship) of x_i. The waving dotted lines indicates undetermined caller-callee relationships. To avoid cluttering the figures we will not put an arrow on these caller-callee relations (solid and waving dotted lines), the top-down hierarchy will make it easy to distinguish caller and callee. Dashed lines with arrows indicate conflict relationship, e.g. x_l conflicts with and precedes y_1.

In the above description of the relationships between x_l and x_r and the y's it was not considered the possibility of an algorithmic precedence of x_l over y_1, or y_n over x_r. If we assume that x_l algorithmically precedes y_1 then such a precedence would be enforced by $p = lca(x_l, y_1)$. Node p is either x_i or one of its proper ancestors. If p is x_i the implicit assumption that the y's, y_1 in particular, are not children of x_i is violated. If p is a proper ancestor of x_i, all the children of x_i have to precede y_1 and our assumption that y_1 precedes x_r would be violated. Similar arguments can be used to show that y_n cannot have an algorithmic precedence over x_r.

Let x and y be conflicting nodes of an execution. Recall that the conflict between x and y is relieved if whenever a descendent x' of x conflicts with a descendent y' of y there is a pair of relieving nodes. We now show that the conflict between x_l and y_1 is relieved. For the sake of contradiction, lets assume that the conflict between x_l and y_1 is not relieved. The conflict between x_l and y_1 originates because of descendents of x_l that conflict with descendents of y_1. Without loss of generality, assume that x_l' and y_1' descendents of x_l and y_1 respectively are the nodes that cause the conflict that is not relieved as shown in Figure 5.4 (a). Let t_x be the child of $lca(x_l, y_1)$ that is also a proper ancestor of x_l (t_x is an ancestor of x_i). Since the conflict is not relieved, the lock request for y_1' will be granted only after the locks of conflicting descendents of x_l are inherited by $lca(x_l, y_1)$ after the termination of t_x. Thus all descendents of t_x precede y_1'. In Figure 5.4 (a) this is indicated by the straight dotted line over x_l' and y_1' and the dotted lines coming from t_x. Since the SLR algorithm always chooses the node whose rightmost child is the leftmost among nodes that have only leaves as children, all the descendents of t_x are separated and reduced before y_1 or its ancestors are separated

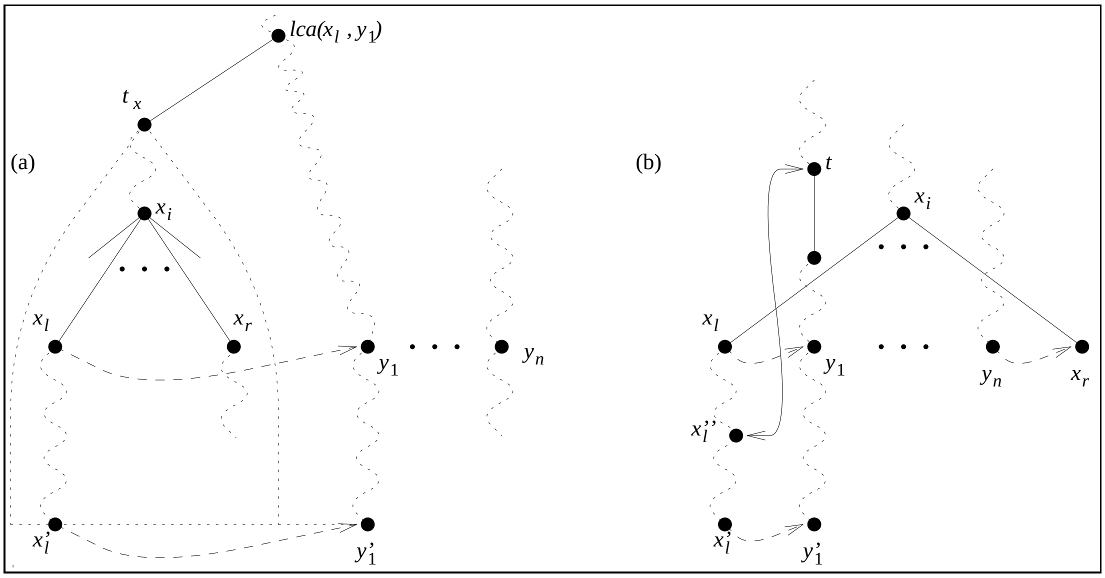

Figure 5.4: (a) Conflict (x_l, y_1) not relieved, (b) conflict (x_l, y_1) relieved.

and reduced. Hence y_1 cannot get interleaved within x_i in execution c_i as assumed originally, a contradiction. Thus, if we want to continue assuming that such a c_i exists we have to assume that the conflict between x_l and y_1 is relieved. Figure 5.4 (b) shows the conflict between x_l' and y_1' descendents of x_l and y_1 respectively, the double arrow solid line indicates that nodes (x_l'' and t) that relieve the conflict.

We now introduce the notion of *clusters* in sequences of y's. The interaction between how the SLR algorithm works and y's that are related by algorithmic precedence renders a peculiar structuring. Consider that y_k algorithmically precedes y_{k+1} as shown in Figure 5.5 (a). We will indicate in the figures that x algorithmically precedes y by connecting the caller-callee edge incident at x to the caller-callee edge incident at y using a dashed line with arrow. Let $t' = lca(y_k, y_{k+1})$ be the node that enforces the algorithmic precedence. Let t_y' be the child of t' that is an ancestor of y_k. All the descendents of t_y' precede the descendents of y_{k+1}. Recall that SLR separates and reduces the nodes whose rightmost child is the leftmost among the nodes that have only leaves as children. Hence, SLR separates and reduces the whole hierarchy of t_y' before considering the separation and reduction of descendents of y_{k+1}. Thus y_k is not simply a descendent of $lca(y_k, y_{k+1})$, y_k must be the same as t_y' as shown in Figure 5.5 (b).

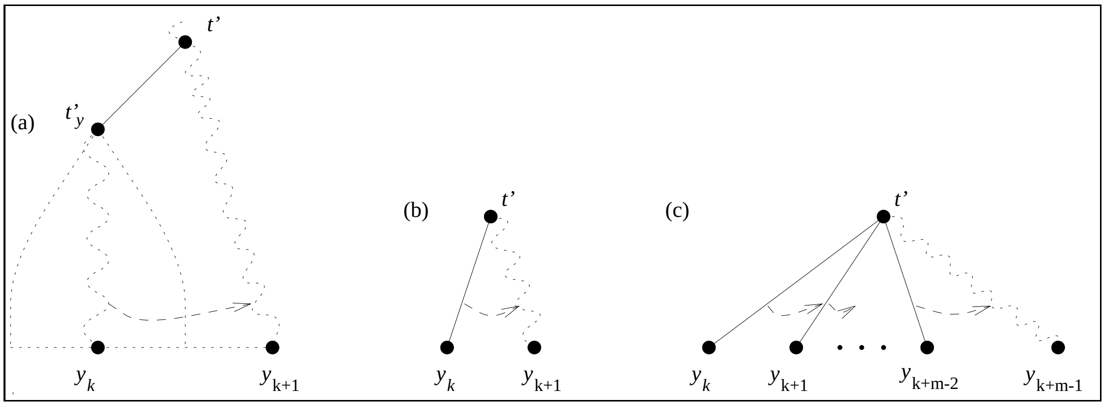

Figure 5.5: (a) Two y's related algorithmically, (b) Two y's related algorithmically in the context of c_i, (c) A cluster.

A maximal set of j contiguous y's related by algorithmic precedence forms a structure where the $j-1$ leftmost of them are siblings. Such a maximal set of contiguous y's are called *clusters*. Figure 5.5 (c) shows an example of a cluster. Observe that the parent of the rightmost node of a cluster must have descendents to the right of x_r, since otherwise SLR would separate and reduce the whole cluster before considering x_i.

We will now show that irrespective of how the y's relate to each other (conflict or algorithmic precedence) a contradiction with respect to the existence of such execution c_i can be derived. We conclude that the conflict between x_l and y_1 is relieved by x_l'' and t. Since x_i is a proper ancestor of x_l'' and t is a proper ancestor of y_1 then x_i is an indirectly commuting ancestor of y_1. We will now show that x_i is an indirectly commuting ancestor of all the y's and then derive a contradiction to the existence of an execution such as c_i.

We assume that x_i is an indirectly commuting ancestor of y_k and we show that x_i is an indirectly commuting ancestor of

- y_{k+1}, when y_k conflicts with y_{k+1}; or
- y_{k+m-1}, when y_k is the leftmost node of a cluster of m y's.

We divide the discussion into 3 cases:

1. y_k conflicts with y_{k+1} and the conflict is not relieved. In this case all the descendents of t_{y_k} the child of $lca(y_k, y_{k+1})$ that is an ancestor of y_k will precede

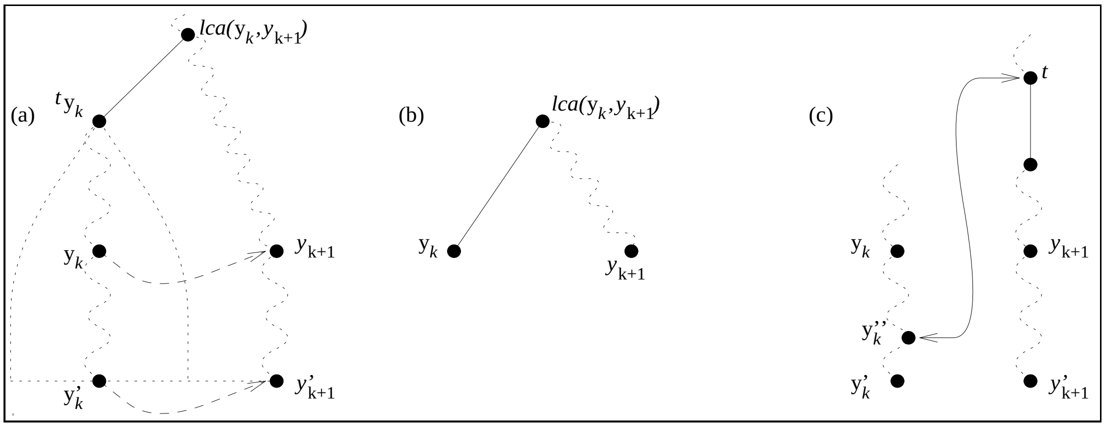

Figure 5.6: Cases where y_k conflicts with y_{k+1}.

a descendent of y_{k+1} as shown in Figure 5.6 (a). SLR will separate and reduce all the descendents of t_{y_k} before considering x_i with child x_r to the right of y_{k+1}. Hence y_k is not simply a descendent of t_{y_k}, y_k must be the same node as t_{y_k} as shown in Figure 5.6 (b). Thus all ancestors of y_k are ancestors of y_{k+1}. Since by assumption x_i is an indirectly commuting ancestor of y_k and any proper ancestor of y_k is a proper ancestor of y_{k+1} then x_i is an indirectly commuting ancestor of y_{k+1}.

2. y_k conflicts with y_{k+1} and the conflict is relieved. Without loss of generality, assume that y'_k and y'_{k+1}, descendents of y_k and y_{k+1} are the nodes that cause the conflict as shown in Figure 5.6 (c). Nodes y''_k, descendent of y_k, and t, proper ancestor of y_{k+1} relieve the conflict. Since x_i is an indirectly commuting ancestor of y_k and a descendent of y_k relieves a conflict together with a proper ancestor of y_{k+1}, then x_i is an indirectly commuting ancestor of y_{k+1}.

3. y_k is the leftmost node of a cluster of m nodes. We refer to a cluster like the one depicted in Figure 5.5 (c). All the proper ancestors of y_k are ancestors of y_{k+m-1}. Since by assumption x_i is an indirectly commuting ancestor of y_k and any proper ancestor of y_k is a proper ancestor of y_{k+m-1} then x_i is an indirectly commuting ancestor of y_{k+m-1}.

Above we were able to show that x_i is an indirectly commuting ancestor of each of the y's, in particular x_i is an indirectly commuting ancestor of y_n. We will now find a contradiction to the assumption that execution c_i can exist. First we present the following proposition.

Proposition 5.1 *Consider a suffix $y_l, y_{l+1}, \ldots, y_n$, $1 \leq l \leq n$, of the y's. Assume that the relationship among nodes of the suffix are algorithmic or non-relieved conflicts, i.e. no nodes that conflict have their conflict relieved. Then all the proper ancestors of y_l are proper ancestors of y_n.*

Proof. Induction on the number c of clusters of nodes related algorithmically in the subsequence (we assume that one node not related algorithmically to others forms a one node cluster). For the basis $c = 1$ it is trivial as depicted in Figure 5.5 (c). As an induction hypothesis assume that the proposition is true for suffixes with $c - 1$ clusters. Let y_m be the rightmost node of the first cluster and y_{m+1} be the leftmost node of the second cluster of a suffix with c clusters. Since y_m and y_{m+1} are not related algorithmically, the relationship between y_m and y_{m+1} must be a non-relieved conflict. The induction hypothesis applies to the suffix starting with node y_{m+1}, and hence all the proper ancestors of y_{m+1} are proper ancestors of y_n. Since the conflict between y_m and y_{m+1} is not relieved the conditions involving y_k and y_{k+1} of Figure 5.6 (a) and (b) apply and hence all the proper ancestors of y_m are ancestors of y_n. Nodes $y_l, y_{l+1}, \ldots, y_m$ form a cluster as the one in Figure 5.5 (c), and hence all the proper ancestors of y_l are proper ancestors of y_m. Thus all the proper ancestors of y_l are ancestors of y_n. $\square_{\text{Proposition}}$

We now analyze the two possibilities with respect to whether the conflict between y_n and x_r was relieved or not.

1. (The conflict between y_n and x_r was not relieved.) Without loss of generality, let y'_n be be one of the descendents of y_n that conflicts with a descendent x'_r of x_r in execution c_1 as shown in Figure 5.7 (a). In this case $lock(x'_r)$ was obtained only after $lock(y'_n)$ was inherited by $t'' = lca(y_n, x_r) = lca(y'_n, x'_r)$. SLR algorithm always separates and reduces the node whose rightmost child is the leftmost among the nodes that have only leaves as children. Thus SLR will separate and reduce x_i only after it has separated and reduced all the descendents of t_{y_n}, the node that is an ancestor of y'_n and a child of t''. Hence y_n is not simply a

descendent of t'', y_n must be the same as t_{y_n} as shown in Figure 5.7 (b). Hence y_n is the child of an ancestor of x'_r, and all the proper ancestors of y_n are ancestors of x_i. Consider the sequence $x_l, y_1, y_2, \ldots, y_n, x_r$ and let p and y_l be contiguous nodes in this sequence such that p is the rightmost element of the sequence that has its conflict with the immediate successor, y_l, relieved. If execution c_i exists such pair exists since x_l and y_1, as we have shown, conflict and their conflict is relieved. From node y_l up to y_n there are only non-relieved conflicts and algorithmic relationships and hence by Proposition 5.1 all proper ancestors of y_l are proper ancestors of y_n. Recall that we have shown that x_i is an indirectly commuting ancestor of all the y's and hence ancestors of x_i are also indirectly commuting ancestors of the y's. The proper ancestor of y_l, t, that relieves the conflict with a descendent p' of p is either a directly or indirectly commuting ancestor of p'. Node t is an ancestor of x_i, and if $p = x_l$, t is a direct commuting ancestor of p', if $p \neq x_l$, t is an indirect commuting ancestor of p'. Thus a violation of rule 3 occurs when we consider that the conflict between p and y_l is relieved, a contradiction that SLR started with an execution produced by semantic locking.

2. (The conflict between y_n and x_r was relieved.) Without loss of generality, let y'_n be a descendent of y_n that conflicts with x'_r, a descendent of x_r, in execution c_1 as shown in Figure 5.7 (c). Consider that the conflict is relieved by y''_n, a descendent of y_n and t'' a proper ancestor of x_r (t'' is an ancestor of x_i). Since x_i is an indirectly commuting ancestor of y_n, and y_n is an ancestor of y''_n then x_i is an indirectly commuting ancestor of y''_n. Since t'' is an ancestor of x_i then t'' is an indirectly commuting ancestor of y''_n, a violation of rule 3 of the protocol.

As shown above it does not matter whether the conflict between y_n and x_r is relieved or not, in both cases we derive contradictions and SLR does not find an execution c_i with a blocking sequence of y's as we assumed. Thus for every execution c_i node x_i can be separated and reduced. $\square_{\text{Lemma}}$

Hence the "Separate and reduce x_i" step can always be executed. Since there are a finite number of nodes and we are reducing them at the end of the Separate and Reduce algorithm we will get a serial execution. Thus semantic locking only produces semantically serializable executions. $\square$

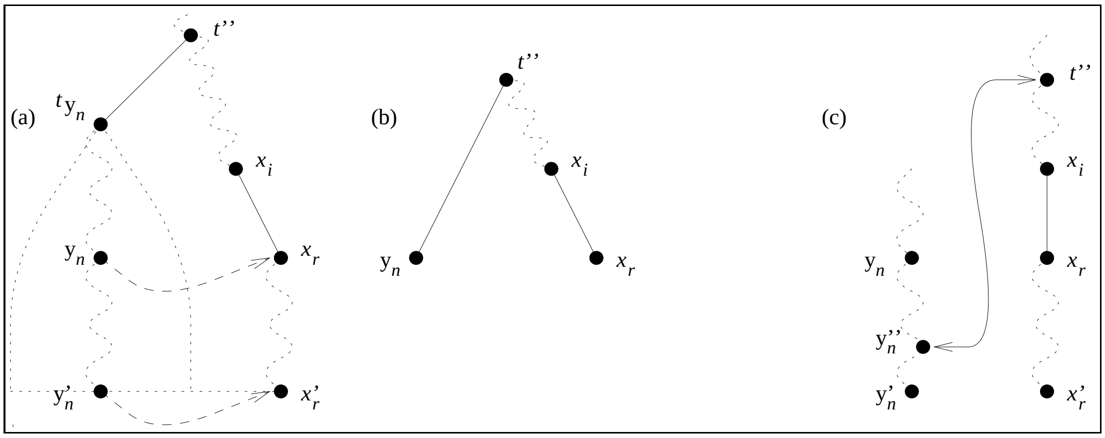

Figure 5.7: (a)&(b) Conflict (y_n, x_r) not relieved, (c) conflict (y_n, x_r) relieved.

5.2.4 Revisiting indirectly commuting ancestors

Figure 5.8 shows an excerpt of an execution where operations a' and b' conflict. Consider that nodes a and b, and nodes a_1 and a_2 commute. Assume that there are no pairs of nodes relieving conflicts, in particular even though a_1 and a_2 commute, they do not relieve conflicts. We cannot use nodes a and b to relieve the conflict between a' and b' because a is an indirectly commuting ancestor of b. This happens because we use the "commute" and "ancestor" relations between method executions to define the "indirectly commuting ancestor" relation, and rule 3 of semantic locking protocol does not allow the use of nodes related by the indirectly commuting ancestor relation.

In reality the fact that a pair of nodes commute should not create a restriction to other pairs. The restriction must be enforced only when a pair that commutes is effectively used by the protocol to relieve a conflict. The refined rule 3 of semantic locking is shown bellow.

Nodes a and b *relieve* a conflict if they are a commuting pair of nodes that comply to rule 3 of semantic locking protocol below.

Node a is an *indirectly relieving ancestor* of b iff there is a sequence of nodes $a_1, a_2, \ldots, a_{2n}$, $n \geq 1$, such that

1. a is a proper ancestor of a_1, and a_1 and a_2 relieve a conflict; and

2. for $1 < k < n$, a_{2k} is a proper ancestor of a_{2k+1}, and a_{2k+1} and a_{2k+2} relieve a

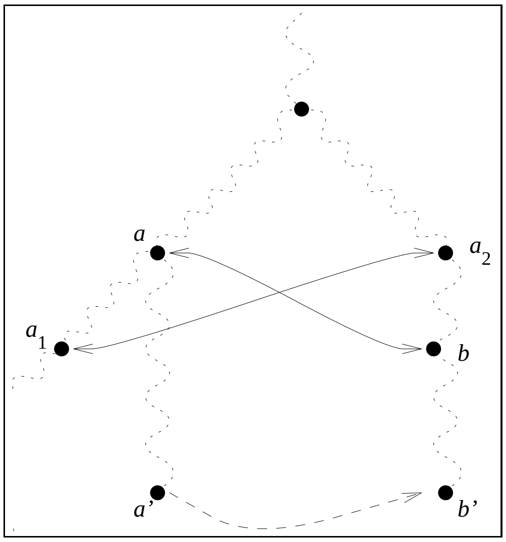

Figure 5.8: Undue restriction on commuting nodes

conflict; and

3. a_{2n} is a proper ancestor of b.

We can also define the above relation inductively as follows. Node a is an *indirectly relieving ancestor* of b iff there exists nodes a_1 and a_2 such that

1. a_1 and a_2 relieve some conflict, a is a proper ancestor of a_1, and b is a proper descendent of b; or

2. a is an ancestor of a_1, b is a descendent of a_2, and a_1 is an indirectly relieving ancestor of a_2.

The new rules of semantic locking are as follows.

1. Same as before.

2. Same as before.

3. A lock $lock(t)$ can be granted to a method execution iff

   (a) No other method execution holds a conflicting lock, and

   (b) All retainers, x'', of a conflicting $lock(x)$ are such that

- x'' is an ancestor of t, or
- x'' is not an ancestor of t, and some element x' of $(x'' \cdots x]$ and some ancestor t' of t commute, where t' is not an ancestor of x' nor an indirectly relieving ancestor.

This redefinition does not brings any impact with respect to the proof of correctness. We presented the concepts in this way to simplify the presentation.

5.3 OH-Semantic Locking Protocol

Semantic Locking needs to keep track of pairs of nodes that were used in relieving conflict information in order to enforce the restriction on the use of indirectly relieving nodes. If the object oriented database can be designed in such way that objects obey a hierarchy then it is not necessary to keep track of the indirectly commutative ancestor relation. The object hierarchy ensures that it will not be possible that indirectly relieving nodes appear. We will now introduce formally the notion of object hierarchy that will enable us to present OH-semantic locking protocol.

We assume that objects are ordered in a hierarchy (object are organized in a partial order) and a method in one object can only call methods on objects that are lower in the hierarchy. More formally, if x is a method execution then $obj(x)$ denotes its corresponding object. If o is an object then $rank(o)$ maps o in the partial order defined by the object hierarchy. The partial order we assumed for objects restricts the way methods can call each other. Formally, let x and x' be method executions, if x is a proper ancestor of x' then $rank(obj(x)) > rank(obj(x'))$. Let x be a method execution, to simplify, we will write $rank(x)$ meaning $rank(obj(x))$.

5.3.1 OH-semantic locking rules

1. A method execution t' can execute an atomic operation t, iff $lock(t)$ is requested and is granted (we say that the method execution t' *holds* $lock(t)$).

2. A method execution cannot terminate (commit or abort) until all its children have terminated. When a method execution terminates:

   (a) *It is not top-level and it commits:* Its locks are inherited by its parent (we say that the parent *retains* the inherited locks).

(b) *It is not top-level and it aborts:* Its locks are discarded.

(c) *It is top-level:* Its locks are discarded.

3. A lock $lock(t)$ can be granted to a method execution iff

   (a) No other method execution holds a conflicting lock, and

   (b) For all other non-ancestor methods x'' that *retain* a conflicting $lock(x)$, some element x' of $(x'' \cdots x]$ and some ancestor t' of t commute ((x', t') relieves the conflict (x, t)).

Conflicts can only be relieved by method executions of the same object, hence if x and y relieve a conflict then $obj(x)$ is the same as $obj(y)$, and $rank(x) = rank(y)$. The OH-semantic locking protocol cannot be used for example with an object oriented database that allows implementations of methods like the one in Figure 4.6. There a method can call another method in the same object and OH-semantic locking assumes a more strict model as discussed above. It is exactly because of this that OH-semantic locking does not need to keep track of relieving information. The object ordering assumption guarantees that there will be no direct commuting nor indirectly relieving nodes.

5.3.2 Proof of correctness

The proof of correctness of OH-semantic locking mimics the proof already presented for semantic locking. It only differs at the final part when considering the relationship between y_n and x_r in the context of the hypothetical sequence of y's that would block the separation of x_i. We will briefly state the key points as proved before and elaborate on this final part.

Theorem 5.2 *OH-semantic locking allows only serializable executions.*

*Proof.*Consider the algorithm SLR as presented in the proof of correctness of semantic-Locking. If this algorithm can separate and reduce all nodes of an execution produced by OH-semantic locking until a serial execution is produced, then OH-semantic locking allows only serializable executions.

Lemma 5.2 *For every c_i, node x_i can be separated and reduced as prescribed by the SLR algorithm.*

Proof. The only reason why x_i in an execution c_i cannot become separated is when there exists a sequence of leaves $y_1, \ldots y_n$ related among them and to two children, x_l and x_r of x_i in the following way:

1. x_l conflicts with and precedes y_1,
2. for $1 \leq j < n$, y_j conflicts with and precedes y_{j+1} or y_j algorithmically precedes y_{j+1}, and
3. y_n conflicts with and precedes x_r.

If such a c_i exists then the conflict between x_l and y_1 was relieved by x_l'' and t as in Figure 5.4 (b). The proof is the same as in the proof of semantic locking.

We will now show that irrespective of how the y's relate to each other (conflict or algorithmic precedence) the relationship in terms of rank of x_i and the y's will enable us to derive a contradiction with respect to the existence of such execution c_i. As stated above, the conflict between x_l and y_1 is relieved by x_l'' and t. Hence $rank(x_l'') = rank(t)$ since commuting methods must be in the same object. But x_i is a proper ancestor of x_l'' thus $rank(x_i) > rank(t)$. Since t is a proper ancestor of y_1, $rank(x_i)$ is greater than the rank of a proper ancestor of y_1. We will now show that $rank(x_i)$ is greater than the rank of all the proper ancestors of the y's and then derive a contradiction to the existence of an execution such as c_i.

We assume that $rank(x_i)$ is greater than the rank of a proper ancestor of y_k and we show that $rank(x_i)$ is greater than the rank of a proper ancestor of

- y_{k+1}, when y_k conflicts with y_{k+1}; or
- y_{k+m-1}, when y_k is the leftmost node of a cluster of m y's.

We divide the discussion in 3 cases:

1. y_k conflicts with y_{k+1} and the conflict is not relieved. In this case all the descendents of the child of $lca(y_k, y_{k+1})$ that is an ancestor of y_k will precede a descendent of y_{k+1} as shown in Figure 5.6 (a). SLR will separate and reduce all the descendents of t_{y_k} before considering x_i with child x_r to the right of y_{k+1}. Hence y_k is not simply a descendent of t_{y_k}, y_k must be the same node as t_{y_k} as shown in Figure 5.6 (b). Thus all ancestors of y_k are ancestors of y_{k+1}. Since by assumption $rank(x_i)$ is greater than a proper ancestor of y_k, $rank(x_i)$ is greater than the rank of a proper ancestor of y_{k+1}.

2. y_k conflicts with y_{k+1} and the conflict is relieved. Without loss of generality, assume that y'_k and y'_{k+1} , descendents of y_k and y_{k+1} are the nodes that cause the conflict as shown in Figure5.6 (c). Nodes y''_k, descendent of y_k, and t, proper ancestor of y_{k+1} relieve the conflict. Since nodes y''_k and t must be in the same object, $rank(y''_k) = rank(t)$. Since y_k is an ancestor of y''_k, $rank(y_k) \geq rank(t)$, and since t is a proper ancestor of y_{k+1}, $rank(y_k) > rank(y_{k+1})$. But $rank(x_i) > rank(y_k)$ and thus $rank(x_i)$ is greater than the rank of a proper ancestor of y_{k+1}.

3. y_k is the leftmost node of a cluster of m nodes. We refer to a cluster like the one depicted in Figure 5.5 (c). All the proper ancestors of y_k are ancestors of y_{k+m-1}. By assumption $rank(x_i)$ is greater than the rank of a proper ancestor of y_k and hence, trivially, $rank(x_i)$ is greater than the rank of a proper ancestor of y_{k+1}.

Above we were able to show that $rank(x_i)$ is greater than the rank of a proper ancestor of each of the y's, in particular $rank(x_i)$ is greater than the rank of a proper ancestor of y_n. We will now find a contradiction to the assumption that execution c_i can exist.

We now analyze the two possibilities with respect to whether the conflict between y_n and x_r was relieved or not. The first case is much more simpler than the corresponding case of semantic lock correctness proof.

1. (The conflict between y_n and x_r was not relieved.) Without loss of generality, let y'_n be be one of the descendents of y_n that conflicts with a descendent x'_r of x_r in execution c_0 as shown in Figure 5.7 (a). In this case $lock(x'_r)$ was obtained only after $lock(y'_n)$ was inherited by $t'' = lca(y_n, x_r) = lca(y'_n, x'_r)$. SLR algorithm always separates and reduces the node whose rightmost child is the leftmost among the nodes that have only leaves as children. Thus SLR will separate and reduce x_i only after it has separated and reduced all the descendents of t_{y_n}, the node that is an ancestor of y'_n and a child of t''. Hence y_n is not simply a descendent of t'', y_n must be the same as t_{y_n} as shown in Figure 5.7 (b). Hence y_n is the child of an ancestor of x'_r, and all the proper ancestors of y_n are ancestors of x_i. But this contradicts our conclusion that $rank(x_i)$ is greater than the rank of a proper ancestor of y_n.

2. (The conflict between y_n and x_r was relieved.) Without loss of generality, let y'_n be a descendent of y_n that conflicts with x'_r, a descendent of x_r, in execution

c_0 as shown in Figure 5.7 (c). Consider that the conflict is relieved by y''_n, a descendent of y_n and t'' a proper ancestor of x_r (t'' is an ancestor of x_i). Hence $rank(y''_n) = rank(t'')$, and since t'' is an ancestor of x_i, $rank(y''_n) \geq rank(x_i)$. Since y_n is an ancestor of y''_n, $rank(y_n) \geq rank(x_i)$. But this contradicts our conclusion that $rank(x_i) > rank(y_n)$.

As shown above it does not matter whether the conflict between y_n and x_r is relieved or not, in both cases we derive contradictions and SLR does not find an execution c_i with a blocking sequence of y's as we assumed. Thus for every execution c_i node x_i can be separated and reduced. $\square_{\text{Lemma}}$

Hence the "Separate and reduce x_i" step can always be executed and at the end of the Separate and Reduce algorithm we will get a serial execution. Thus OH-semantic locking only produces semantically serializable executions. $\square$

5.4 Other works

In traditional databases, the most commonly adopted correctness criterion is serializability. Object-oriented database systems such as ORION [GK88, Kim90] and O_2 [CF90, BDK92] ensure the serializable execution of transactions. Both of these database systems employ a locking protocol based on multigranularity locking [Gra78]. Multigranularity locking was developed to reduce locking overhead by organizing the database in the form of a hierarchy, e.g., the database is composed of files, and each file is composed of a set of records etc. By obtaining a lock at a higher level, transactions implicitly lock entities at the lower levels. The designers of ORION and O_2 developed similar protocols replacing the database hierarchy by the class and object hierarchy in object oriented databases. However, their approach is based on the traditional notion of *read* and *write* locks and the strict two phase locking protocol. Thus, these approaches do not employ the semantics of higher level operations available in object-oriented databases. Furthermore if an object is locked, every descendant of that object in the hierarchy is also locked even if the transaction may only access a small subset of that sub-hierarchy. In particular, a significant drawback with this approach is the following. In object-oriented databases, a transaction execution begins by executing an operation or a method on a higher level object which in turn invokes operations or methods on other objects resulting in a nested structure of the transaction execution.

By using the standard two-phase protocol with multigranularity locking, ORION and O_2 do not exploit the nested behavior of transaction executions to increase concurrency. However, the major strength of these two systems, from the transaction management point of view, is that they directly address the issue of schema evolution and extensibility. In particular, the proposed protocols allow concurrency between schema changes as well as user transactions. This approach was also applied to complex objects with pre-declaration of locks [HDK$^+$90].

We have shown a problem with the protocol of [MRW$^+$93]. Other aspects of the protocol have potential problems. This protocol requires locks not only for the atomic operations but also for method executions. In particular, the system designer has to derive a conflict or commutativity table not only for the atomic operations of each class but also for all the method executions of all the classes in the database. It is also interesting to notice that to enhance concurrency it is necessary to allow more interleaving. This protocol is motivated by increasing concurrency by the use of semantics, even if it is fixed, has the potential of decreasing concurrency. Consider top-level method executions that have operations conflicting with operations of other method executions. If simply one of these conflicts is not relieved then their corresponding top-level methods cannot be executed in an interleaved manner, potentially defeating the goal of more concurrency.

Agrawal and El Abbadi [AE92] have proposed a concurrency control protocol for providing transactional access to object-oriented databases. They provided a uniform treatment for dealing with both class and instance objects and extend the two phase locking protocol to objects with arbitrary operations. The novel aspect of this protocol is that conflicting operations may share locks on the same objects, provided that the order between the operations is preserved. This protocol is extended for synchronizing nested executions of transactions on objects. Allowing updates to both classes and instances of these classes makes the proposed protocol beneficial for extensibility and experimentation, which are needed for continuously evolving object oriented database applications. However, this protocol also does not exploit the semantics of higher level methods to increase concurrency.

Chapter 6

Conclusion

We have presented a conflict serializability theory for nested transactions where definitions and concepts match with each other to provide a balanced framework. This statement is supported by the fact that we were the first to prove necessary and sufficient conditions for conflict serializability of nested transactions. Our contribution characterizes the whole class of nested transaction executions serializable by a conflict predicate on the atomic operations.

We have presented locking protocols for nested transactions in the context of object oriented database. Object Oriented databases not only give the motivation for the use of nested transactions, we hope that they also will provide semantic information that is *effective* in enhancing concurrency. Our semantic locking protocols are, as far as we know, the first protocols that produce semantically serializable executions and have a proof of correctness. Traditional database applications with simple "flat" transactions are well served by traditional concurrency control protocols. Our protocols give one direction in extending database capabilities in terms of more complex and structured transactions.

6.1 Future Work

There is a significative number of opportunities where semantic knowledge can be used to enhance concurrency. One important work is to investigate some groups of applications and check how easily such commutativity information can be found.

We have investigated only the issues related to serializability. A fundamental investigation is related to the feasibility of our assumption that aborted transactions have no effects in the database.

The simulation parameters related to flat transactions are well understood. However simulation parameters for nested transactions are almost non-existent in the literature. This is in part due to the fact that only recently commercial systems that use nested transactions have started to appear. Some examples of commercial systems are Encina [She93b, She93a] and Kala [SG91]. It seems that these systems do not use semantic information to enhance concurrency but they claim the use of nested transactions techniques. Beyond the investigation of opportunities for the use of semantics as mentioned above is the investigation of simulation parameters for nested transactions. If the investigations above prove to be fruitful one important future work is the determination of parameters that are important in the performance of systems driven by ours protocols. Another future work to formalize more powerful proof techniques for semantic serializability, possibly graph-based, that would permit easier correctness proofs. The style of our correctness proofs gives important directions for such endeavor.

Bibliography

[AE92] D. Agrawal and A. El Abbadi. A Non-restrictive Concurrency Control Protocol for Object Oriented Databases. In *Proceedings of the Third International Conference on Extending Data Base Technology (EDBT'92)*, volume 580 of *Lecture Notes in Computer Science*, pages 469–482. Springer-Verlag, March 1992. An extended version of this paper will appear in Distributed and Parallel Databases: An International Journal.

[BBG89] C. Beeri, P. A. Bernstein, and N. Goodman. A Model for Concurrency in Nested Transactions Systems. *Journal of the ACM*, 36(2):230–269, April 1989.

[BDK92] F. Bancilhon, C. Dolobel, and P. Kanellakis, editors. *Building an Object-Oriented database System: The story of O_2*. Morgan Kaufmann, 1992.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, Reading, Massachusetts, 1987.

[CF90] M. Cart and J. Ferrié. Integrating Concurrency Control into an Object-Oriented Database System. In *Proceedings of the International Conference on Extending Data Base Technology, Springer-Verlag, LNCS 416*, pages 363–376, 1990.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, November 1976.

[FLMW90] A. Fekete, N. Lynch, M. Merrit, and W. Weihl. Commutativity-based locking for nested transactions. *Journal of Computer and System Sciences*, 41(1):65–156, August 1990.

[FLW90] A. Fekete, N. Lynch, and W. Weihl. A Serialization Graph Construction for Nested Transactions. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 94–108, April 1990.

[GK88] J. F. Garza and W. Kim. Transaction Management in an Object-oriented Data Model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 37–55, June 88.

[Gra78] J. N. Gray. Notes on database systems. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer-Verlag, 1978.

[HDK+90] U. Herrmann, P. Dadam, K. Kuspert, E. Roman, and G. Schl ageter. A Lock Technique for Disjoint and Non-Disjoint Complex Objects. In *Proceedings of the International Conference on Extending Data Base Technology, Springer-Verlag, LNCS 416*, pages 219–237, March 1990.

[HH91] T. Hadzilacos and V. Hadzilacos. Transaction Synchronization in Object Bases. *Journal of Computer and System Sciences*, 43(1):2–24, August 1991. Special issue on the 7th Annual ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems, March 21-23, 1988.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[HR93] Theo Härder and Kurt Rothermel. Concurrency control issues in nested transactions. *iThe VLDB Journal*, 2(1):39–74, January 1993.

[HW91] M. P. Herlihy and W. E. Weihl. Hybrid Concurrency Control for Abstract Data Types. *Journal of Computer and System Sciences*, 43(1):25–61, August 1991. Special issue on the 7th Annual ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems, March 21-23, 1988.

[Jag90] T. Jagodits. Concurrency control in nested transactions: A new protocol and its comparative performance evaluation. Master's thesis, University of California Santa Barbara, 1990.

[Kim90] Won Kim. *Introduction to Object-Oriented Databases.* The MIT Press, 1990.

[Kor83] H. F. Korth. Locking primitives in a database system. *Journal of the ACM*, 30(1):55–79, January 1983.

[Kri93] Vasudha Krishnaswamy. *Semantics Based Synchronization in Database Systems.* PhD thesis, University of California Santa Barbara, 1993.

[Mos81] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, Massachusetts Institute of Technology, 1981.

[Mos85] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* MIT Press, Cambridge, Massachusetts, 1985.

[MRW+93] P. Muth, T. C. Rakow, G. Weikum, P. Brössler, and C. Hasse. Semantic Concurrency Control in Object-Oriented Database Systems. In *Proceedings of the 9th IEEE International Conference on Data Engineering*, pages 233–242, April 1993.

[Pap86] C. H. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, 1986.

[RA92] Rodolfo F. Resende and Amr El Abbadi. A graph testing concurrency control protocol for object bases. In *Proceedings of the 4th International Conference on Computing and Information*, pages 316–319, May 1992.

[Ree78] D. P. Reed. *Naming and Synchronization in a Decentralized Computer System.* PhD thesis, Massachusetts Institute of Technology, 1978.

[SG91] S. S. Simmel and I. Godard. The kala basket: a semantic primitive unifying object transactions, access control, versions, and configurations. In *Object-Oriented Programming Systems, Languages, and Applications*, pages 230–246, October 1991.

[She93a] M. Sherman. Architecture of the encina distributed transaction processing family. In *SIGMOD Record*, pages 460–463, June 1993.

[She93b] M. Sherman. Distributed transaction processing with encina. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 268–269, January 1993.

[Wei89] W. E. Weihl. The impact of recovery on concurrency control. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, pages 259–269, March 1989.

[Wei91] G. Weikum. Principles and realization strategies of multilevel transaction management. *ACM Transactions On Database Systems*, 16(1):132–180, 1991.

[Won93] Man Hon Wong. *Context-Based Synchronization in Object-Oriented Databases*. PhD thesis, University of California Santa Barbara, 1993.