Concurrency Control
in
Object–Based Systems

by

Michael Edward Zapp

A thesis
presented to the University of Manitoba
in partial fulfilment of the
requirements for the degree of
Master's of Science
in
Computer Science

Winnipeg, Manitoba, Canada, 1993

ISBN 0-315-85969-5

Canada

Name _____

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Computer Science
SUBJECT TERM

0984 U·M·I
SUBJECT CODE

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION
General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523
Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS
Language
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291

Literature
- General 0401
- Classical 0294
- Comparative 0295
- Medieval 0297
- Modern 0298
- African 0316
- American 0591
- Asian 0305
- Canadian (English) 0352
- Canadian (French) 0355
- English 0593
- Germanic 0311
- Latin American 0312
- Middle Eastern 0315
- Romance 0313
- Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY
Philosophy 0422
Religion
- General 0318
- Biblical Studies 0321
- Clergy 0319
- History of 0320
- Philosophy of 0322

Theology 0469

SOCIAL SCIENCES
American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce-Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- Ancient 0579
- Medieval 0581
- Modern 0582
- Black 0328
- African 0331
- Asia, Australia and Oceania 0332
- Canadian 0334
- European 0335
- Latin American 0336
- Middle Eastern 0333
- United States 0337

History of Science 0585
Law 0398
Political Science
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Plant Physiology 0817
- Range Management 0777
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Physiology 0433
- Radiation 0821
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

EARTH SCIENCES
Biogeochemistry 0425
Geochemistry 0996
Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES
Environmental Sciences 0768
Health Sciences
- General 0566
- Audiology 0300
- Chemotherapy 0992
- Dentistry 0567
- Education 0350
- Hospital Management 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology 0380
- Occupational Health and Therapy 0354
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Physical Therapy 0382
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES
Pure Sciences
Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Electronics and Electricity 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756
- Solid State 0611

Statistics 0463

Applied Sciences
Applied Mechanics 0346
Computer Science 0984
Engineering
- General 0537
- Aerospace 0538
- Agricultural 0539
- Automotive 0540
- Biomedical 0541
- Chemical 0542
- Civil 0543
- Electronics and Electrical 0544
- Heat and Thermodynamics 0348
- Hydraulic 0545
- Industrial 0546
- Marine 0547
- Materials Science 0794
- Mechanical 0548
- Metallurgy 0743
- Mining 0551
- Nuclear 0552
- Packaging 0549
- Petroleum 0765
- Sanitary and Municipal 0554
- System Science 0790

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY
General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

CONCURRENCY CONTROL IN OBJECT-BASED SYSTEMS

BY

MICHAEL EDWARD ZAPP

A Thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Abstract

The object-oriented paradigm is an important area of research not only in programming languages but also in databases and operating systems. Databases and operating systems require concurrent access to objects. This thesis provides a model of concurrent access to objects, based on the concept of transactions. It rigorously formulates the different types of transactions that participate on an object–based system and the types of histories (or logs) that must be maintained. A definition of object serializability (called O-serializability) is introduced and a graph theoretic tool is described that defines precisely when a given execution sequence is serializable. An architecture for transaction management in an object–based system is defined to describe the interactions between the different software components of the transaction management facilities. The architecture is used to identify the components that are required for concurrency control. This thesis introduces algorithms for these components, defines the interactions between the components, and the protocols required to ensure serializability. The model and graph theoretic tool is used to show the correctness of the algorithms in ensuring O-serializability. The algorithms maintain serializability with respect to the objects and the user's transactions in a semi-autonomous manner through the use of a global correctness algorithm and algorithms within the objects.

Acknowledgements

There are many people that I would like to thank for their support and encouragement. First and foremost I wish to thank Dr. Ken Barker. The many hours of discussions made this thesis possible. I would also like to thank him for allowing me to pursue this topic, when I asked for a challenging area of research, that's exactly what he gave me. Secondly, I would like to thank my peers in the Advanced Database Systems Laboratory. The insights attained through our many "discussions" have proved valuable both in this thesis and in many other facets of life. I would also like to thank Jasmine for her many hours of companionship while I was writing this thesis. Last, but not least, I wish to thank my wife Tracy. Your support and encouragement over the last two years has been invaluable. I love you and look forward to the many challenges and adventures we will face in the future.

Contents

List of Figures

Chapter 1

Introduction

Recent database research has focused on moving away from the relational data model. Advances in applications require the use of complex modeling techniques that do not exist in relational database systems. Many of these needs can be met by *object-oriented systems*, which define abstract representations of entities in an environment. There is a large commercial demand for object-oriented database systems (or simply *object–based systems*) that provide for the rich modeling capabilities while providing the functionalities of traditional database systems (such as transaction management, query processing, and version management).

This thesis addresses the problem of concurrent access to objects within an object base. When multiple users access a database, their accesses must be controlled so that anomalies such as lost updates and the reading of inconsistent data do not occur. This is traditionally done through the use of *transactions*. Transactions are units of work that are *atomic*, *consistent*, *isolated*, and *durable*. These are known as a transaction's ACID properties because each leaves a consistent database in a "new" consistent state after its completion. A transaction is not aware

of any other executing transactions, has all of its updates made permanent and recoverable, and is a single unit of work such that all or none of its operations occur. This thesis is concerned with the support of atomicity and isolation during the concurrent execution of transactions on an object base.

Correctness of the concurrent execution of a set of transactions is defined through *serializability*, which means that the transactions execute in an equivalent to serial manner. This notion of serializability is extended for object bases using a definition of *object-serializability*. This provides a theoretical basis for the discussion of concurrency control algorithms.

The nature of objects complicate concurrency control in object–based systems. An object allows for abstractions above the internal implementation of its functionalities through the encapsulation of data and the means of accessing the data. This structuring is used to define two scheduling algorithms, one for internal object executions and another of the executions of the transactions that access the object base. Object-serializability is the correctness criterion from which these algorithms are shown to be correct.

Early work in object bases has concentrated on developing prototypes that provide for the storage of persistent objects [20], object-oriented design through the storage of a schema and schema evolution [8], and the development of queries for execution on objects [20]. Transaction management has not been studied at great lengths in these systems. Traditional techniques have been applied at the object level so that studies of the other areas of database systems can proceed. A summary of existing work is given in Chapter 2. This thesis adds to this work by introducing a new means of discussing serializability in object bases and a new suite of concurrency control algorithms that exploit the added functionality that exists in object–based systems.

1.1 Object Orientation

Before discussing transaction management and concurrency control it is useful to introduce the concepts of object orientation. Since this thesis does not address all of the requirements of object-oriented databases, only the basic properties are discussed. Definitions of the more complex modeling techniques can be found in Barker *et al.* [3].

Object orientation stems from the use of abstract data types. These types provide for the encapsulation of data and the procedures that manipulate the data and require that the types only be accessed via the procedures. In object orientation, the data are *attributes* and the procedures are *methods*. Objects are instantiations of the type definitions which have values for the attributes and can accept method invocations.

Object orientation expands on this by defining complex modeling techniques for defining types. The type definitions are called *classes* with an object being the instance of a particular class. A class hierarchy is created with a root class from which all other classes descend. A class can *inherit* definitions from its ancestors. Attribute definitions can be inherited and modified by a subclass. Methods can be inherited and used or modified by a subclass. A class does not have to inherit all definitions from it ancestors, only those that are relevant to the definition of the class.

These and other modeling tools provide advanced means for designing software and data. Concurrency control, however, only deals with the execution of programs not their design. Therefore, issues such as inheritance do not affect the design of concurrency control algorithms. They provide effective tools in the development of advanced applications but only the active objects within an object–based

system are affected by concurrency control.

1.2 An Architectural Framework

Before presenting the remainder of the thesis, this section presents a transaction management architecture for object–based systems. The architecture gives an abstract view of the software components that facilitate transaction management. The functionalities and high-level interactions of the components are discussed. This provides a framework for transaction management in object–based systems and allows for discussions of the components and their requirements.

Most of the work in the area of architectures for transaction management exists in the realm of classical databases for centralized and distributed systems. The architecture presented here exhibits the same functionality described by Bernstein *et al.* [5] in centralized database systems and adapts some of the structure found in the distributed architecture presented by Özsu and Valduriez [23].

An overview of the two main components of the architecture is presented, followed by a detailed description of each. The overall model is illustrated in Figure 1.1. The object–based management system contains an *Execution Monitor* and an *Object Processor*. The Execution Monitor (discussed in Section 1.2.1) receives user transactions and returns the results. Its purpose is to provide an interface to users and to coordinate and schedule the method invocations on behalf of user transactions.

When the Execution Monitor wishes to schedule a method, it submits the method to the Object Processor (discussed in Section 1.2.2) for execution. The object processor schedules and executes individual method operations. When the

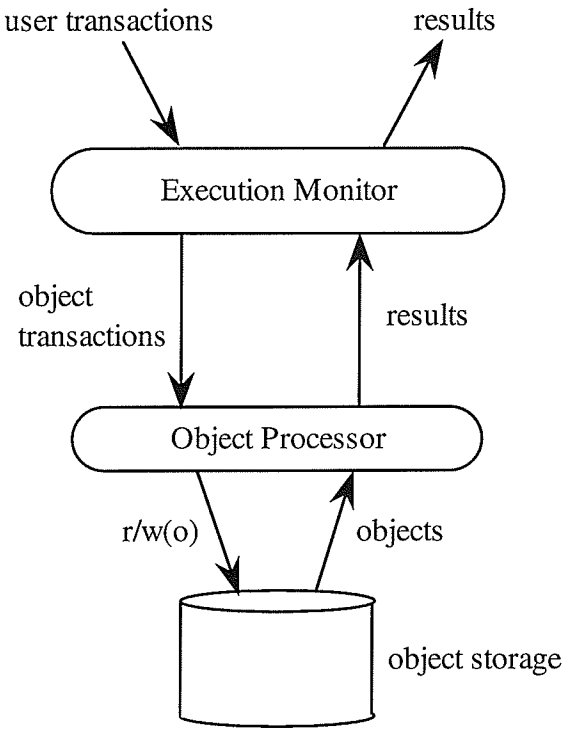

Figure 1.1: Centralized Architecture Overview

method completes it returns the result to the Execution Monitor. Method results may vary depending on the object model and the transaction model being enforced. For example, if a nested transaction model is used intermediate results and a "prepare-to-commit" may be returned while an open nested model may return final results and a commit or an abort notification. The essential architectural feature is the overall protocol required between the Execution Monitor and the Object Processor. In processing method executions, the Object Processor will retrieve and update objects by accessing the *object storage*. The object storage is a stable storage used for keeping permanent copies of the objects. It takes read and write requests on objects. A write causes the object storage to overwrite the

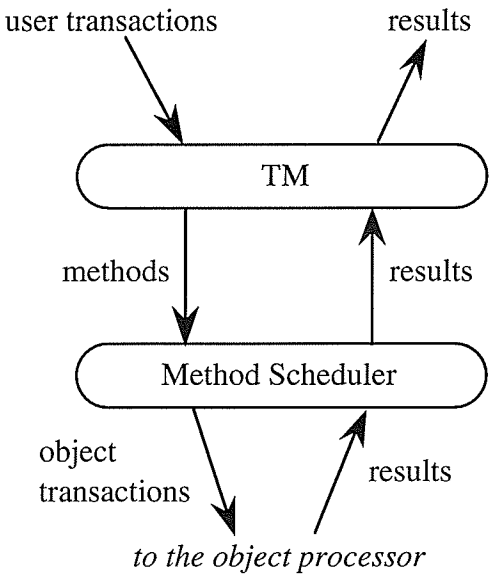

Figure 1.2: The Execution Monitor

current copy of the object with the new copy given to it.[1] This facilitates a commit decision made by the Object Processor. When the object storage receives a read request it will return the required object to the Object Processor.

1.2.1 The Execution Monitor

The Execution Monitor architecture is depicted in Figure 1.2. The *Transaction Manager* (TM) receives user transactions and coordinates their execution. It acts as the external interface to the other components of the object–based system. The Transaction Manager sends the invocation requests to the *Method Scheduler*. The final task of the Transaction Manager is to return results to the user.

[1]Selecting the update inplace approach is made to simplify the overall architecture but alternate approaches may be more desirable when addressing reliability and implementation issues. At this point the FIX/FLUSH [23] decisions are seen as orthogonal.

The Method Scheduler receives requests for method invocations from the Transaction Manager and returns the results to the Transaction Manager upon completion of the method. Its purpose is to implement an inter-object concurrency control algorithm. The algorithm synchronizes the execution of all user transactions in the system by properly interleaving the executions of their methods. The method execution *per se* is the responsibility of the Object Processor.

The Execution Monitor is responsible for ensuring that execution orderings affecting different objects are handled correctly. The orderings within an object must be compatible with the orderings of other objects, with respect to the user transactions, for inter-object correctness to be maintained. This does not require a particular algorithm to maintain conflict serializability within each object but rather that an algorithm at the Execution Monitor level maintains consistency between objects. Therefore, the execution of operations at objects is orthogonal to the design of the Execution Monitor.

1.2.2 The Object Processor

Figure 1.3 illustrates the architecture of the Object Processor. It receives object transactions and passes them to the *Object Manager*. The Object Manager (OM) takes method invocations and returns their results to the Execution Monitor. Since our model permits each object to use the concurrency control algorithm of its choice, it is necessary for the Object Manager to determine the *Object Scheduler* that should receive the method invocation. The Object Manager accomplishes this by mapping objects to corresponding schedulers using its local object dictionary. The Object Manager takes the method invocation and passes it to the necessary Object Scheduler.

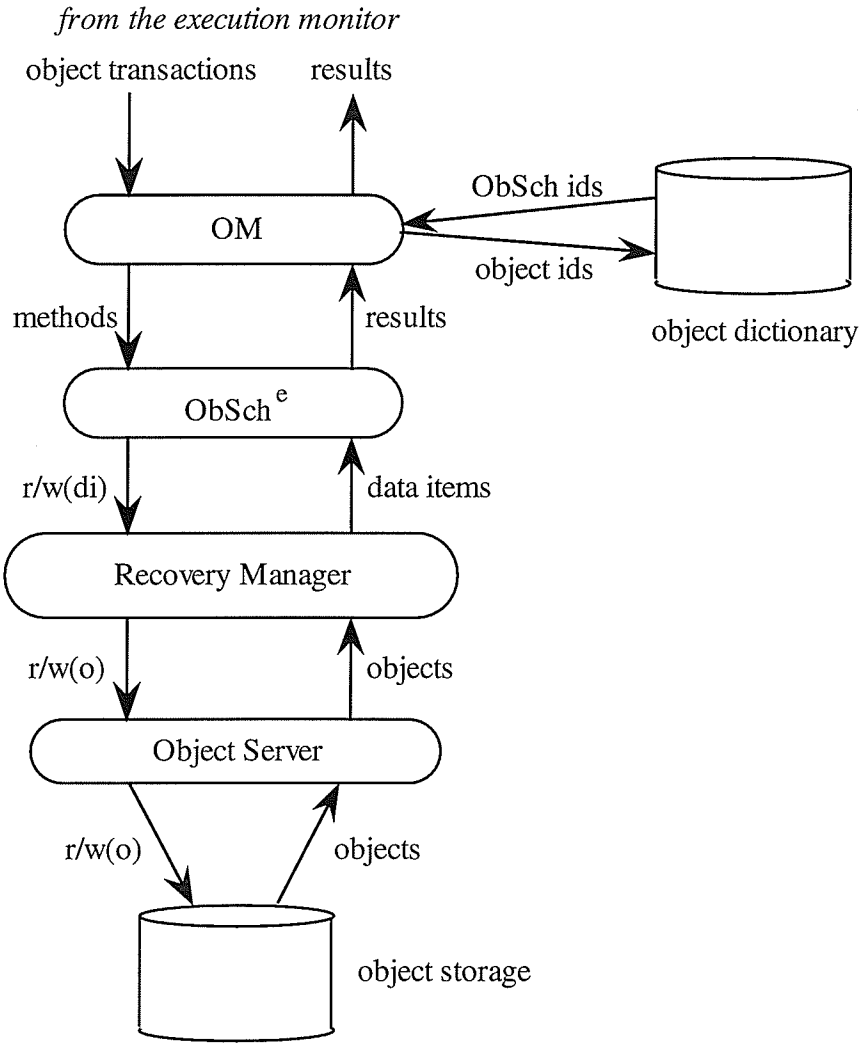

Figure 1.3: The Object Processor

There is a logical Object Scheduler for each object (we identify the scheduler for object e by ObSch^e) which implements a concurrency control algorithm for its object. The algorithm synchronizes the execution of methods submitted to that object. When a method completes, the Object Scheduler will return the result to the Object Manager.

An Object Scheduler accesses an object's data using traditional transaction read (r) and write (w) operations on data items. The *Recovery Manager* receives requests for data and ultimately returns the requested data item. The Recovery Manager implements the reliability protocols necessary for the objects to maintain consistency by guaranteeing that the objects will return to, or remain in, a consistent state following a failure. When a commit is performed, the Recovery Manager ensures that all updates made by the committing transaction are permanent. It also ensures that when an abort takes places, none of the updates made by the aborting transaction are permanent or visible.

The Recovery Manager executes in consort with the *Object Server*. The Object Server reads and writes objects to and from the object storage. This is analogous to the Cache Manager in a classical database system [5] reading and writing pages of data for its Recovery Manager. It may however, due to performance considerations, only do so for the subset of an object that is actually accessed. It takes requests from the Recovery Manager to read or write a specific object and sends the specific request to the object storage. The Object Server takes an object from storage and places it in volatile memory where it can be accessed by the Recovery Manager to fulfill requests from the Object Schedulers. It also takes objects in volatile memory and writes them to the object storage to make updates to the objects permanent. The Object Server will only perform these actions upon specific instruction from the Recovery Manager. The Recovery Manager must make

"fix and flush" decisions (as described in Özsu and Valduriez [23] for distributed systems) with respect to the movement of objects to and from stable storage. The Object Server implements these decisions through its management of object storage.

1.2.3 Discussion

This architecture provides a framework for the model presented in this thesis and a means for arguing its potential applicability. It allows each object to use its own concurrency control algorithm for maintaining serializability. This yields greater flexibility in using optimized algorithms for the data structures and needs of each particular object (observed by Hadzilacos and Hadzilacos [13] to be a useful attribute for concurrency control in object bases). It also provides for a level of abstraction above reads and writes on data items. Users can create transactions that only invoke high level operations, thus making the design of transactions more intuitive and less prone to error.

The presented architecture does not, however, impose a particular transaction model on the object bases. Models for abstract data types, such as the model introduced by Weihl and Liskov [30], can make use of this architecture through its abstraction above reads and writes. The architecture also provides for much more. It allows for the support of more complex objects than those based on abstract data types. The notion of object orientation discussed in Section 1.1 can make use of this architecture. In the object-oriented environment an object may need the services of other objects, thus requiring it to invoke methods on other objects. This architecture facilitates this by allowing a method to invoke other methods through the use of the Execution Monitor (to be described in Chapter 4). This also enables the use of active objects; objects that will cause other actions to be performed

based on some result. These actions occur without the knowledge of the user who submitted the initial request. This architecture models this by the invocation of a method which subsequently invokes other methods, possibly on other objects, to accomplish the necessary actions.

The model presented in Chapter 3 attempts to utilize all of the functionality provided by the architecture. It defines serializability with respect to each object and the entire object base, thereby permitting different algorithms at each object, under the coordination of a global algorithm, to maintain serializability over the entire object base. It also defines a nesting structure for the methods of the objects. The model allows for a method to invoke other methods, possibly on different objects, so that object hierarchies and active objects can be supported. This thesis provides the correctness criterion and algorithms for the Method Scheduler and Object Schedulers of this architecture. Algorithms and theory for the other components of the architecture, aside from the intuitive descriptions of their behaviour given above, are left as open problems.

1.3 Outline of Thesis

The remainder of the thesis proceeds as follows. Chapter 2 defines the traditional transaction model and discusses previous related work. A formal transaction model allows for the study of serializability in an object–based system. Both of these are discussed in Chapter 3. Chapter 4 discusses the issue of concurrency control through the introduction of algorithms for object and user level scheduling. Finally, Chapter 5 makes some concluding remarks and identifies directions for future research.

Chapter 2

Background

A survey of both motivational and related work aids in the understanding of the model and algorithm defined in this thesis. The motivational work details research done in isolation of object bases but is applicable to object bases. The related work is a summary of research on the topic of concurrency control in object bases.

This chapter has the following structure. Section 2.1 gives a summary of traditional transactions and concurrency control algorithms. This provides a basis from which the rest of the thesis can draw. Section 2.2 summarizes the research that provides the motivation for this thesis. Finally, Section 2.3 describes the models, concurrency control algorithms, and prototypes that exist for object bases.

2.1 Traditional Transactions

Traditional transaction management defines a simple transaction model whereby each transaction consists of a sequence of reads and writes. The execution of a transaction on a consistent database will leave the database in a new consistent

state. Concurrency control is then the interleaving of multiple transactions such that they will leave the database in a consistent state following their execution. Many algorithms exist for both centralized and distributed databases based on this traditional model. This work is summarized in Bernstein *et al.* [5] and Papadimitriou [24]. To distinguish transactions defined in this model from those to be discussed later, these transactions are identified as *flat transactions*. The remainder of this section defines flat transactions and the traditional correctness criterion and discusses some concurrency control algorithms. This section provides a means of progressing towards a model for object bases while avoiding issues that are not directly relevant to the thesis.

Flat Transactions

Flat transactions were first defined for classical databases and are well understood. The definitions for classical transaction management are taken from Özsu and Valduriez [23] and form a framework for subsequent definitions. We begin with some nomenclature. $O_{ip} \in$\{read,write\} is operation p of transaction i. OS_i is the set of all operations of transaction i, and we say $OS_i = \bigcup_k O_{ik}$. For transaction i we have a termination condition $N_i \in$\{commit,abort\}. Thus the definition of a flat transaction is as follows:

Definition 2.1.1 A flat transaction T_i is a partial order $T_i = (\Sigma_i, \prec_i)$, where

1. $\Sigma_i = OS_i \bigcup \{N_i\}$,
2. for any two $O_{ip}, O_{iq} \in OS_i$, if $O_{ip} = r(x)$ and $O_{iq} = w(x)$ for any x, $O_{ip} \prec_i O_{iq}$ or $O_{iq} \prec_i O_{ip}$, and
3. $\forall O_{ip} \in OS_i$, $O_{ip} \prec_i N_i$. ■

Point (1) is obvious. Point (2) states that if any two operations within the transaction conflict (in that one is a read and the other a write on the same data item), then an ordering for those two operations must exist in $\prec_i$. Point (3) ensures that all operations of a transaction occur before the termination of the transaction.

Histories

A log or *history* records the execution of transactions. Histories provide a means of analyzing the execution of transactions in order to ensure that they have a correct ordering when executed concurrently. The definition of a history for flat transactions is as follows:

Definition 2.1.2 Given a set of transactions $\mathcal{T}$, a history (H) is a partial order $H = (\Sigma, \prec)$, where

1. $\Sigma = \bigcup_j \Sigma_j$ where Σ_j is the domain of transaction $t_j \in \mathcal{T}$,
2. $\prec \supseteq \bigcup_j \prec_j$ where $\prec_j$ is the ordering relation for transaction T_j, and
3. for any two conflicting operations $p, q \in H$, either $p \prec q$ or $q \prec p$. ■

This definition implies that if the operations of two transactions conflict, the transactions are similarly in conflict. Transactions and their histories enable the discussion of the correctness of transaction execution, through the notion that these transactions may conflict.

Conflict Serializability

A correctness criterion for flat transactions is *conflict serializability*, whereby the conflicting operations of two transactions must be ordered such that the transactions appear to execute serially (ie. all the operations of one appear to execute

before all the operations of the other). This correctness criterion is enabled through the definition of serial histories and the equivalence of histories, as follows:

Definition 2.1.3 A history $H = \{T_1, \ldots, T_n\}$ is *serial* iff $(\exists p \in T_i, \exists q \in T_j$ such that $p \prec q) \Rightarrow (\forall r \in T_i, \forall s \in T_j, r \prec s)$. ■

Definition 2.1.4 Two histories are *conflict equivalent* if they are over the same set of transactions and they order conflicting operations in the same way. ■

These definitions enable the definition of a serializable history, yielding a correctness criterion for transactions. The definition of a serializable history is as follows:

Definition 2.1.5 A history is *serializable* if and only if it is equivalent to a serial history. ■

From this correctness criterion concurrency control algorithms can be shown to be correct. If a concurrency control algorithm only produces equivalent to serial executions of transactions, it is correct. This is proven through the use of *serialization graphs* that are defined as follows (taken from Bernstein *et al.* [5]):

Definition 2.1.6 The serialization graph for a history H is a directed graph such that the vertices are the committed transactions in H and the edges are all $T_i \rightarrow T_j (i \neq j)$ such that an operation of T_i precedes and conflicts with an operation of T_j in H. ■

The ability to use such graphs is proven by the *serializability theorem*:

Theorem 2.1.1 A history is serializable iff its corresponding serialization graph is acyclic.

Proof See Bernstein *et al.* [5, page 33]. ■

A history is serializable if and only if its serialization graph is acyclic. This implies that a concurrency control algorithm is correct if and only if all of the histories generated by it have acyclic graphs.

View Serializability

Another notion of correctness is *view serializability*. It differs from conflict serializability in the definition of the equivalence of two histories. Two histories are not equivalent if they order conflicting operations similarly, but are equivalent when they have the same reads-from relations. Each read operation reads its value from the last operation that wrote the value. Therefore, two histories are equivalent if their read operations have the same view by reading from the same write operations. Papadimitriou [24] formally defines this correctness criterion. Papadimitriou also shows that the problem of demonstrating that a given history is view serializable is NP-complete. For this reason, conflict serializability is the correctness criterion most often selected for transaction models.

Concurrency Control Algorithms

Many algorithms have been developed and proven correct according to the transaction model based on conflict serializability. They can be broadly classified into one of two categories: optimistic and pessimistic. Pessimistic algorithms delay conflicting operations when first identified while optimistic algorithms allow all operations to proceed and verify their correct execution once a transaction completes. Along with these there are also hybrid algorithms that combine features of both pessimistic and optimistic algorithms.

Two-Phase Locking

The most common form of a pessimistic algorithm is the two-phase locking algorithm. This algorithm works as follows. When a transaction wishes to access a data item it must first obtain the appropriate lock (either a read or write lock depending on whether the operation is a read or a write, respectively). Two locks are in conflict if their corresponding operations are conflicting, as defined in Definition 2.1.1 for reads and writes. If another transaction holds a conflicting lock, it must wait for that transaction to release its lock before obtaining the lock. A transaction can obtain locks until it releases a lock, at which point it can no longer obtain any locks. This results in the two phases, a lock request phase followed by a lock release phase. A strict form of this forces a transaction to retain all of its locks until it completes, releasing them immediately before the transaction's termination.

Two-phase locking can be shown to be correct through a contradiction. If two-phase locking were not correct, a cycle would exist in a serialization graph for some history produced by the algorithm. Suppose such a graph exists with a cycle between two transactions T_1 and T_2. This implies an edge from T_1 to T_2 and an edge from T_2 to T_1. The first edge arises from T_1 releasing a lock on a data item which is subsequently locked in a conflicting mode by T_2. The second edge arises from T_2 releasing a lock on a data item which is subsequently locked in a conflicting mode by T_1. This cannot occur with two-phase locking since the transactions would be allowed to obtain locks after they have released a lock for this cycle to be generated. Induction can be used to show that this is true given a cycle containing any number of transactions. Therefore, two-phase locking produces only conflict serializable schedules.

Optimistic Certification

Optimistic algorithms take the form of *certifiers* [5]. These algorithms retain a

history of all operations executed by all transactions. When a transaction commits, a serialization graph is constructed based on the history. If the graph is acyclic, the execution was correct and the updates of the transaction are made permanent. If the graph is not acyclic, all of the transaction's operations are rolled back and the transaction is resubmitted. These algorithms have been proven conflict serializable. Since they test for the acyclicity of the serialization graph in their algorithms and abort transactions that cause cycles, certifiers allow only serializable executions. Optimistic concurrency control suffers from increased overhead due to the abortion and resubmission of transactions.

Two-Phase Commit

When a transaction executes in a distributed environment it is split into a set of transactions, one for each site that the transaction executes on. A protocol is required for the transaction to commit atomically. Such a protocol is the *two-phase commit* protocol [5]. It requires communication between the sites so that all sites come to the same termination decision for the transaction. The site where the transaction begins is the *coordinator* and all other sites are the *participants*.

The protocol proceeds as follows. After the coordinator has sent transactions to the participants for execution it waits for replies. After completion of a transaction, a participant sends its termination decision to the coordinator. If the decision is to abort it aborts the transaction. Otherwise, it waits. Once the coordinator has received all replies it makes a decision. If any reply is an abort, the decision is to abort and it sends an abort message to all the participants. Otherwise, it sends a commit to all the participants. The participants, upon receiving the message, carry out the specified action. This protocol is of particular importance to the algorithms presented in Chapter 4 of this thesis.

Summary

This fundamental model is the basis from which more complex models have been defined. It is the starting point for the models discussed in the subsequent sections and in the model defined in this thesis. The concepts introduced by this model form the basis for research on transaction management in databases. The terminology introduced above will be used throughout this thesis and knowledge of this basic model is assumed in the discussion of the models and algorithms introduced below in subsequent sections.

2.2 Motivation

There are two areas of research that motivate this thesis: namely, nested transactions and concurrency control in abstract data types. Nested transactions are relevant since the invocation of methods by other methods leads to a natural nested structure. Abstract data types are the precursors to objects and are therefore important in introducing some of the complexities that arise when considering objects. This section first discusses abstract data types and concludes with a survey of nested transaction models and a discussion of their relevance to object–based systems.

2.2.1 Abstract Data Types

Abstract data types are similar to classes in that they both specify the attribute types and operations for their instance objects. Each abstract data type has local data that can only be accessed through the operations defined for the abstract data type. Much research has attempted to define models and algorithms for the concur-

rent execution of transactions invoking the operations of abstract data types. Much of the work has been done by Weihl and others [30, 28, 29, 17]. Other contributions have been made by Herlihy [15, 16], Schwarz and Spector [27], and Badrinath and Ramamritham [2]. The following will describe the model of correctness normally associated with abstract data types and look at some of the proposed concurrency control algorithms.

Any correctness criterion must ensure that concurrently executing transactions appear atomic and isolated. Such a model has been defined for transactions on abstract data types. The model defines concurrency control with respect to each abstract data type. Each abstract data type has a local algorithm to ensure the correct execution of its local operations. To enable this, each transaction is *fully blocking*, which means that when a transaction invokes an operation it cannot invoke another operation until the first returns. This guarantees that an ordering of operations at an abstract data type will be the same as the ordering of their invoking transactions. In addition, Weihl [28] defines that each operation of an abstract data type is *atomic*. Therefore, the execution of the operations of an abstract data type cannot by interleaved.

Three types (or properties) of local atomicity are defined for abstract data types: static, dynamic, and hybrid. Correct executions are attained if all abstract data types within a system use the same atomicity property thereby making global correctness achievable. Weihl [29] has shown that all abstract data types in a system must use the same atomicity property because different properties will create incompatible orderings at different abstract data types, which would eliminate the possibility of global correctness.

Each of the atomicity properties relates to the type of algorithms that can be used to achieve that property. Static atomicity occurs when predefined in-

formation of transactions exist for ensuring transaction serializability. The most common manifestation of this is through the use of timestamps where each transaction obtains a unique timestamp when it begins execution and is the basis of correctness when the transaction invokes operations. Dynamic atomicity uses information attained through the execution of transactions to determine an ordering that is serializable. The most common dynamic protocol is two-phase locking where locks are attained as the transactions execute, not before the execution begins. Hybrid atomicity is a combination of the previous properties, using both static and dynamic information to ensure a serializable ordering of the transactions.

Once an atomicity property is chosen it is necessary to provide a concurrency control algorithm with information about the operations of the abstract data type. The algorithm will then be able to correctly schedule the execution of the operations for the transactions that invoke operations on the abstract data type. A *serial specification* accomplishes this by detailing the proper execution of each operation of the abstract data type (through pre- and post-conditions) when all transactions execute serially. This serial specification is given by the developer who designs and implements the abstract data type. The serial specification can be described as a compatibility matrix (for example see Badrinath and Ramamritham [2]) defining those operations in the abstract data type that commute and conflict. The concurrency control algorithm uses this information to ensure that conflicting operations of the abstract data type are ordered in a consistent manner; based on its atomicity property.

Unlike models which define commutativity as the results of two operations being the same regardless of their order [2], Weihl [28] defines two types of commutativity: *forward commutativity* and *backward commutativity*. These commutativity relations are defined with respect to the type of recovery mechanism used and the

implications they have on concurrency. Forward commutativity is based on *intention lists* where a transaction writes all of its updates in a list and applies them to the database when it commits. This implies that transactions are always reading a committed (and consistent) database. Since all updates are applied to the database at commit time, none of the updates can leave the database in an inconsistent state (which would violate the semantics of a transaction). Therefore, forward commutativity is defined to ensure that this will not occur by considering undefined states in the definition of commuting and conflicting operations.

Backward commutativity, on the other hand, is based on *undo logs* where updates are made directly to the database and the operations are stored in a log. If a transaction aborts, the operations are undone by scanning the log and applying the inverse of the operations. In this case, updates are not necessarily applied to a consistent database but the database is ensured to be consistent when the transaction commits. Therefore an undefined state is permissible provided that the transaction returns the database to a defined state before committing. Thus, backward commutativity does not consider undefined states in determining the commutativity of operations.

This model has resulted in the definition of algorithms for the different atomicity properties. Herlihy [16] and Herlihy and Weihl [17] define algorithms based on hybrid atomicity while Weihl [28] defines algorithms based on dynamic atomicity. Herlihy [15] also defines a multiversion time-stamping algorithm based on static atomicity. All the algorithms presented have been proven correct based on their atomicity property and provide solutions to the concurrency control problem with respect to abstract data types.

There are two broad categories of concurrency control algorithms; they are either optimistic or pessimistic. This is also true for algorithms based on abstract

data types. Two algorithms proposed for abstract data types demonstrate this. The algorithm introduced by Weihl [28] is a pessimistic algorithm using locking. This locking is based on which operations of an abstract data type conflict, as given in the serial specification. It is similar to two-phase locking except that a transaction is blocked from invoking operations if it currently has an operation pending. An optimistic algorithm was introduced by Herlihy [16]. In this algorithm, the execution of transactions proceed, according to the model described above, until a transaction commits. At this point validation proceeds and unless the execution was correct the transaction is aborted. Validation is based on the atomicity property employed (hybrid in this case), using the information attained to analyze the execution. This algorithm uses a combination of static and run-time information, in the form of timestamps and optimistic locks, to ensure that the transactions executed correctly.

Concurrency control in abstract data types is a well studied field yielding many useful results. When considering these results with respect to objects, however, some new problems present themselves. These problems arise due to the active nature of objects, in that, the method of an object can invoke a method of another object. This is not defined for abstract data types and the means of defining serial specifications with this additional property seems non-trivial. The abstract data type concurrency control model was not designed to handle such complexities. Therefore, other transaction models must be analyzed to find techniques suitable when methods invoke other methods.

2.2.2 Nested Transactions

When one object's method invokes another, the invocations are said to be *nested*. Therefore, we investigate nested transaction models to identify solutions that can be applied to object bases. Fekete *et al.* [9] introduced two ways of viewing nested transactions: namely, as *data* and *procedural* abstractions. Procedural abstractions were originally proposed as nested transactions by Moss [21]. Data abstractions, also known as multilevel transactions, are another form of nested transactions as defined by Weikum [31, 32] and Beeri *et al.* [4]. The fundamental concept underlying nested transactions is that one may invoke others, which may in turn invoke others, to some, possibly unbounded, depth. Two types of nested transaction models have been defined: namely, open and closed. The work of Moss and Beeri *et al.* is on the closed nested model. We present an intuitive description of closed nesting and defer the formal definitions until the next chapter.

The semantics of closed nesting using procedural abstraction is as follows. A transaction (called a parent) can invoke any number of sub-transactions (called children). Children can be active simultaneously. A transaction with no parent is a top-level transaction and it along with its descendants is a transaction family. The results of any transaction in a transaction family are not made visible to any other transaction family until the top-level transaction for the family completes. Within a transaction family, only leaf transactions can directly access data.

The fundamental principle of closed nesting is that no partial results of any transaction family can be made visible to any other transaction family. The following semantics accomplish this closed nesting. When a child commits all scheduling information pertaining to it is given to its parent. The effects of the commit of a child are only made permanent and visible when its top-level transaction commits.

The abort of a child results in the removal of all of its scheduling information. If a system failure occurs at any time, even after a child has "committed", before a top-level transaction completes, the entire transaction family aborts. Finally, a parent cannot commit until all of its children complete, but it may abort at any time.

Moss [21] defined a two-phase locking concurrency control algorithm for this model. When a transaction accesses a data item, it must acquire the appropriate lock. If another transaction holds a conflicting lock, then the transaction must wait for the release of the conflicting lock. Nesting requires additional rules. When a sub-transaction commits, its parent inherits its locks. When a sub-transaction aborts, its locks are released but any locks on the same data items held by its ancestors are kept by those ancestors. The acquisition of locks is also affected by nesting. A transaction can acquire a write lock only if all other transactions holding write locks on the same data item are ancestors of the transaction. Also, a transaction can acquire a read lock only if all other transactions holding write locks on the same data item are ancestors of the transaction. Moss [21] has shown that this algorithm is correct according to the model of nested transactions discussed above.

The use of data abstraction in nested transactions is discussed in Beeri *et al.* [4] and Weikum [31, 32]. Beeri *et al.*'s work describes a criterion necessary for closed nested transactions to be serializable. They define a group of transaction families to be a computational forest such that the forest must have a correct ordering at all levels of the forest. The model implies that each level above the leaves is an abstraction of the operations performed by the leaf transactions. This requires a restriction: the nesting has a predefined and fixed depth. For the purpose of this thesis, the most important correctness condition of data abstractions is *downward*

order compatibility. Downward order compatibility means that the ordering of two parent transactions must be adhered to by all of their descendants. Therefore, the ordering defined on two children cannot be incompatible with the ordering already defined on their parents. Commutativity is another condition used by the model as a means of rearranging transactions at a given level so that serial orderings can be derived in proofs of correctness. Beeri *et al.* [4] proves the correctness of schedules that adhere to these nested transaction conditions.

Weikum [31] introduced a concurrency control protocol for his model of multilevel transactions. This protocol implements a level-by-level strategy whereby each of the levels in the multilevel system have their own scheduler for ensuring the preservation of conflict serializability at that level. This allows for a modular design where different algorithms can be used at different levels, provided that they ensure conflict serializability. This is a top-down design, a scheduler at a higher level orders transactions according to its view of conflicts which reduces the possible execution orders at lower levels.

Based on this protocol, Weikum [31] introduced a system based on (strict) two-phase locking. As this algorithm ensures conflict serializability, it can be applied at each level of the system. At the bottom level, read and write locks on data items are acquired as in traditional transactions. At higher levels there are operation specific locks such that locks are compatible if and only if their operations are not conflicting, as was discussed for abstract data types in Section 2.2.1. Using these locks, the two-phase locking algorithm executes as in traditional systems. The only difference is in the release of locks. Consider a transaction at a given level invoking a transaction at the level below it. The invoked transaction acquires locks during its execution. These locks are subsequently released when the invoking transaction completes, not when the invoked transaction completes. Thus,

with minor modifications two-phase locking can be used in multilevel transaction systems.

In addition to multilevel transactions, Weikum [32] discusses the general notion of open nesting. With open nesting, when any transaction commits, including sub-transactions, its updates are made visible. This is the main difference between open and closed nesting. Open nesting allows partial results of a transaction family to be made visible to other transaction families. If a parent transaction should abort then all updates must be removed, including those made by children that have committed. The execution of transaction families is serialized in that they must execute in a correct order relative to the top-level transactions. The difficulty with this model occurs because updates performed by sub-transactions are made visible before their parents commit. Thus if a parent should abort after a child has already committed, the updates of the child must be undone. Since the child has committed, other transaction may have seen its updates and must also be aborted. The problem then becomes how the effects of the children are undone and how to deal with cascading aborts. Much research remains to be done to find methods of dealing with these complexities. One proposal, introduced by Garcia-Molina and Salem [10] and adapted by Weikum [32] for the multilevel open nested transaction model, is to use *compensating transactions* to undo a child transaction's actions. Compensating transactions are defined as a part of *Sagas* [10], which provides a solution to the problem of long-lived transactions on multidatabases through the use of open nesting. The complexities of the open nested model and the need for further research were determining factors in the decision to restrict the research focus of this thesis to the closed nested model.

Summary

This thesis only addresses the problem of applying closed nesting using pro-

cedural abstraction to objects. We will, however, use results from nesting using data abstraction that are applicable to the research where appropriate. Unfortunately, problems arise when applying the nested model to objects. Two restrictions are problematic. First, nesting only to a specific level is unrealistic when addressing objects. Second, having only leaf transactions accessing data is impractical with respect to object bases. In the object model, any method on any object can access data and it is unrealistic to predetermine how many levels of nesting will occur when methods are invoking each other. Therefore, one must transform the model for nested transactions into a model viable in the object paradigm by removing these two restrictions. Two such models have been introduced, one using the closed nested model and the other using the open nested model (to be discussed in Section 2.3.2).

2.3 Related Work

This section details research specific to object bases. It begins with a review of prototype object bases from a transaction management point of view in Section 2.3.1. Section 2.3.2 follows with a detailed look at two transaction models for object bases. Finally, Section 2.3.3 describes concurrency control algorithms based on these models.

2.3.1 Prototypes

Most research on object bases has focused on data modeling and query processing. Several prototypes have been developed and discussed in the literature (for example, see [20, 18, 8, 33]). Very little research has directly addressed transaction

management issues in these systems or they make enabling assumptions so their research in data models and query processing can be undertaken.

Most prototype object-oriented database systems can be placed into one of two categories. One category includes systems that have object-oriented interfaces built on top of a traditional database system (typically relational). An example of such a system is Iris [33]. Such systems use the transaction management facilities that are provided by the traditional system and is therefore not the focus of this thesis.

The other category of prototypes is that of systems that build the management system based on the existence of persistent objects, such as ORION [20], Cactis [18], and O_2 [8]. Most of these systems implement transaction management by using object or page level two-phase locking. Although this is comparatively easy to implement it severely restricts concurrency among user transactions and eliminates concurrency possible when multiple transactions must access the same object. Page level locking does not increase the complexity of transaction management as compared to object level locking but allows for the locking of multiple objects on the same page or the locking of parts of an object stored on multiple pages. Cactis' approach uses two-phase locking on the attributes of the objects in its object base. This does not increase the complexity over other prototypes but uses a lower level of lock granularity.

The ORION prototype's transaction management facilities are described by Garza and Kim [11]. ORION, like the other prototypes, uses a two-phase locking protocol on its objects. It also provides additional facilities to address the problem of long-lived transactions in object bases. Long-lived transactions execute over a relatively long period of time, thereby locking out other transactions that require some of the objects held by the long-lived transactions. Garza and Kim [11] pro-

pose two methods to alleviate the delays incurred by these transactions: *private databases* and *hypothetical transactions*. First, a private database causes the objects within it to be inaccessible to all but its defining transaction. This ensures that other transactions cannot be delayed by the long-lived transaction owning the private database. Second, hypothetical transactions never commit and will never change an object. In order to reduce delays the objects accessed by a hypothetical transaction are copied and accessed freely by that transaction. Therefore, no other transactions can be delayed by it because they would access the original objects. We do not address the problem of long-lived transactions in this thesis. Our approach is to rigorously define the transaction and object models and provide insights into "traditional" concurrency control.

2.3.2 Transaction Models

Hadzilacos and Hadzilacos [13] introduce a model for transactions in object bases using closed nesting. In this model, object methods have *local* and *message steps*. Local steps access the object's attributes while message steps invoke methods on other objects. Transactions submitted by users are considered methods of a special system object which only contain message steps. The execution of a method is a transaction on its object and is a *method execution*, which is a partial order of its steps based on the commutativity of those steps. The system must synchronize multiple method executions at each object and ensure that the orderings at each object are compatible with one another.

Hadzilacos and Hadzilacos propose the following model. A history is defined over the set of all method executions on the objects of the object base. The history is a partial order of the method executions based on their internal partial

orders and the conditions necessary to enforce inter-object compatibility. The conditions state that there must be no recursive relations between descendants[1] and that the ordering of method executions and their descendants must be compatible. This compatibility is slightly different from the nested model in that parents and descendants may be on different objects. The correctness of a history is based on its equivalence to a serial history composed of the same method executions and parent–child relations. View equivalence is the basis for equivalence specific to their model. Thus the method executions must have the same views in both the generated history and a serial history for the histories to be equivalent and therefore, correct.

A graph is constructed based on the partial ordering that, if acyclic, is equivalent to some serial ordering. The vertices of the graph are the method executions that appear in the corresponding history. An edge is added between two vertices if an ordering exists between the operations, at the method executions' common ancestor, that (possibly indirectly) invoked the methods. An edge is also added if the two methods have descendants which have local steps that conflict.[2] In both cases, the direction of the edge corresponds to the ordering of the operations. The model guarantees that if this graph is acyclic, the history from which it was derived is (view) serializable.

One important feature of object bases is their ability to have different concurrency control algorithms for each object and have another distinct algorithm ensure their compatibility. Hadzilacos and Hadzilacos' model does not capture this but it models the entire computation of the object base. Therefore, the model is extended to give a more suitable correctness criterion through the reorganization of

[1]That is, a forest results with the roots being the transactions submitted by the users.

[2]Note that in this model the set of descendants of a method execution includes itself.

the graph [13]. The graph derived from the overall history is split into two graphs for each object, one for local operations and the other for the message steps which relate an object with others. If all graphs representing every history are acyclic then the overall history, and hence the global ordering, is serializable. The feature of allowing different concurrency control algorithms for each object is attained by having an algorithm at each object to enforce the acyclicity of the graph for each object's local steps and an algorithm to enforce the acyclicity of the graphs relating the method executions of different objects. This will ensure that the synchronizations at the different objects are correct. Hadzilacos and Hadzilacos point out that this separation has not been realized for their model.

Another, open nested, transaction model for objects has been introduced by Rakow *et al.* [25]. The model defines a set of objects that contain *actions* (also known as methods). The concept of commutativity introduced by Weihl and Liskov [30] is used to maintain an ordering of the actions of an object. Actions execute atomically and may invoke atomic actions on other objects. Transactions submitted by users are top-level transactions that are actions on a special system object. The set of actions that invoke actions on a given object is the set of transactions on that object. Once again, orderings must be made compatible between objects. If there is an ordering of two actions on an object, then any actions invoked by them must reflect that ordering. In addition, the ordering of invoked actions must be reflected in the ordering of the invoking actions.[3]

This model defines a schedule at each object. Correctness is shown by ensuring that each schedule is (conflict) equivalent to a serial schedule (at the action level) and all inter-object relations are conflict serializable. This is done by creating

[3]This must be done explicitly due to the open nesting; with closed nesting this second ordering is guaranteed.

three types of dependencies: transaction, action, and added dependencies. These dependencies are applied at each object. Two transactions on an object have a dependency if their actions conflict or they have sub-transactions that have a dependency. A sub-transaction is an action that a transaction has invoked which is itself a transaction on another object. Two actions have a dependency if they have a semantically defined ordering at their object or they are transactions on another object with a defined transaction dependency for them. This requires that an object shares its transaction dependency relations with the objects that access it. One final dependency handles the case when two actions on different objects invoke actions on the same object. If the accessed object creates a transaction dependency between the two actions, an added dependency relation for the two actions is given to each of the invoking objects.

Rakow *et al.* state that if each object schedule is equivalent to a serial schedule by having the same transaction dependency relations and acyclic action and added dependency relations, global correctness is maintained and the schedules are serializable. This model is different from the model introduced in this thesis in that it uses open nesting. Open nesting is more general than closed nesting but we wish to ensure that transactions submitted by users are completely atomic.

A protocol based on locking is introduced by Muth *et al.* [22] to implement this open nested model. Locks are obtained for the actions of the objects and blocking is enforced using the commutativity relation for the actions of an object. To enforce correctness between objects *retained locks* are defined. These locks are held after a sub-transaction commits and are released when its top-level transaction terminates. This ensures that sub-transactions of other top-level transactions that conflict with the retained lock cannot invoke the action and cause an inconsistent ordering of the top-level transactions. The use of open nesting in the transaction

model requires the introduction of retained locks.

In addition to these two models, the multilevel transaction model is used by Cart *et al.* [7] for the purpose of handling complex objects. The concept of complex objects is an extension of the object-oriented model. It enables more complex data abstractions by constructing objects from other objects. An object can have as a part of its structure and behaviour the structure and behaviour of other objects. This means that parts of an object can be other objects, accessible as if their components were part of the structure and behaviour of the object.

These complex objects form a data abstraction which makes nesting through data abstraction an obvious choice for transaction management in these systems. The model and protocols introduced by Weikum [31] can be used in order to enable complex objects. This model is, however, restrictive as a means of general transaction management in object–based systems.[4] Thus, the observations made by Cart *et al.* [7] in the use of multilevel transactions for complex objects are useful in defining the requirements of transaction models handling complex objects. This thesis does not consider complex objects in the design of the transaction model and concurrency control algorithms.

2.3.3 Concurrency Control Algorithms

Agrawal and El Abbadi [1] define a concurrency control algorithm based on the model introduced by Hadzilacos and Hadzilacos [13]. This is a locking algorithm that uses *ordered sharing* to relate locks. Ordered sharing proceeds as follows. Operations never wait, after executing they obtain either a shared or ordered shared lock relative to each operation that is currently executing. If the completed opera-

[4]These restrictions were identified for nested transaction models in general in Section 2.2.2.

tion commutes with a given active operation, the two operations do not conflict and the completed operation obtains a shared lock with respect to the active operation. If this is not true, the completed operation conflicts with the active operation and obtains an ordered shared lock with respect to the active operation. This implies that the ordering of these operations is important and all other conflicting operations between their invoking transactions must have the same execution order. With respect to object bases, the notion of commuting operations follows from Hadzilacos and Hadzilacos' model discussed above where local and message steps are said to commute or conflict based on their semantic definitions.

This algorithm provides for the nesting of methods and uses a strict two-phase locking protocol based on ordered sharing. The execution of a transaction must adhere to the following rules in order to be considered correct. A lock is associated with each operation of the transaction and has shared or ordered shared relations with all other operations based on the description given above. The transaction holds all locks until it terminates, thus enforcing strict two-phase locking. The transaction cannot terminate until all of its children have terminated. The transaction must adhere to the *ordered commitment rule*, which states that the transaction is waiting for another transaction if it has obtained an ordered shared lock with respect to a lock held by the other transaction and both transactions have a common parent. Recall from the model defined by Hadzilacos and Hadzilacos that there is a special object from which all transactions are invoked, thus causing all transactions within the object base to have a common parent. This implies that the transaction cannot commit until all transactions it is waiting for have terminated. Finally, the *lock inheritance rule* must be adhered to by the transaction. This rule states that when a transaction commits, all of its locks are inherited by its parent. This implies that a transaction's locks and the locks of all of its descendants are inherited by

the transaction's parent. When the system object inherits the locks, implying that the top-level transaction for a given transaction family has committed, it discards the locks.

Another algorithm based on the Hadzilacos and Hadzilacos model has been introduced by Resende and El Abbadi [26]. It employs an optimistic certifier that constructs a serialization graph for each method execution that is invoked. The graph initially has no vertices and is constructed as follows. Recall the serialization graph introduced by Hadzilacos and Hadzilacos [13] that contains vertices for method executions. This algorithm constructs a graph by adding a vertex for each method invoked by a method execution.

Edges are added between nodes of a graph when a method execution completes. An edge is added from any other method that has terminated to this method if there is an ordering defined between the operations that initiated the two methods. An edge is also added from any method that has terminated to this method if they or their descendants have a conflict relation between local steps at an object. These constructions correspond to the two means of adding edges to the serialization graph of the Hadzilacos and Hadzilacos model discussed above.

Following the inclusion of the edges the algorithm tests the graph for cycles. If a cycle exists, the top-level transaction of the method execution's transaction family is aborted. Otherwise, the transaction and its method executions continue. The homogeneity of having user's transactions execute as methods of a system object results in a graph with the vertices being the user's transactions. Therefore, the algorithm is correct with respect to the Hadzilacos and Hadzilacos model since all graphs are acyclic and the combination of the graphs of all the method executions will result in the serialization graph.

Chapter 3

Transactions and Serializability

Recall that a transaction is a unit of work performed on some system. A transaction manager maintains correct, possibly concurrent, executions of these transactions. Transactions are most often interleaved so they appear atomic during concurrent execution [5, 24]. Extending this to object–based systems requires substantial effort, but before achieving advances in this new environment a well-defined model of correctness is needed.

Objects have both behaviour and attributes. Behaviour is implemented with procedures called *methods*. Access to attributes is only by calls to methods. Our model captures the logical separation of the execution of method operations within objects from the execution of transaction methods submitted by users. The model defines what a correct execution sequence of user transactions with respect to each other is and what constitutes a correct execution of method operations within the objects themselves. Therefore, user submitted transactions must be synchronized with respect to each other at both the *user transaction* level and at the level of the *object transactions* executing on their behalf. This chapter presents the definition

of histories for both levels as well as the notion of serializability for object bases.

This chapter introduces a new transaction model and correctness criterion for object–based systems. The model uses conflicts at the operation level of objects to define correct executions within each object, which allows the use of existing algorithms that ensure conflict serializability within the objects. The model links to existing technology by using similar concurrency control concepts thereby making the transition from traditional techniques to those introduced in this chapter easier. The use of conflicts between operations and a logical nested structure for defining the interrelations between objects makes this model immediately applicable when defining concurrency control algorithms that maximize the potential concurrency within objects.

The balance of this chapter is presented as follows. Section 3.1 presents the definition of transactions in this model. Using the model defined in Section 3.1 we discuss the concept of histories in object bases in Section 3.2. The definition of histories allows for the discussion of the the serialization of such histories in Section 3.3. Section 3.4 uses the notion of serializable histories to present a serializability theorem suitable for this model.

3.1 Transactions and The Object Model

This section defines transactions and objects in an object–based system. It begins by giving a simple definition of an object for the purposes of this thesis. Following this a set of detailed definitions of the transactions that exist in the model is presented.

We employ a simple view of objects by restricting the object definition to the aspects necessary for transaction management. This permits us to concentrate

on the important issues and ignore the irrelevant details that would unnecessarily complicate the design (eg. inheritance hierarchies). Object models and related issues are discussed in Kim [19] and Bertino and Martino [6].

An object–based system contains a set of objects that are uniquely identifiable and contain structural and behavioural components. The structural component is a set of uniquely identifiable data items or attributes defined by the structure. They are uniquely identifiable within the context of their containing object. The behavioural component is a set of procedures, usually called methods, that are the only means of accessing the structural components. We define an object as follows.

Definition 3.1.1 An *object* $o = (e, S, M)$ where:

1. e is the unique identifier of o,
2. S is the object's structure; composed of identifiable attributes such that $\forall a_i, a_j \in S,\ i \neq j \Rightarrow a_i \neq a_j$, and
3. M is the object's behaviour; composed of identifiable methods such that $\forall m_i, m_j \in M,\ i \neq j \Rightarrow m_i \neq m_j$. ■

This definition provides the distinction between the passive data and the active behaviour of objects inherent in object–based systems. Point (1) allows unique identification of each object in the environment, and we adapt the following additional notation: o^e identifies object e and the structure and behaviour for e by S^e and M^e, respectively. Point (2) represents the attributes of objects. Point (3) captures the behavioural or active aspects but makes no distinction between a method's identification and its name. A fully implemented system would require method identification using both its name and parameter set, if polymorphism is

to be permitted, but these issues are orthogonal and unnecessarily complicate the model.

Transactions are the fundamental unit of computation in databases and we carry this philosophy forward to this new environment by defining them in object bases. Before describing the approach a feasible alternative is considered. A transaction model could be envisioned that has a single level where a user writes a transaction that invokes a method on an object and returns immediately. Subsequent method invocations execute serially and no invoked method would be able to call other methods. Unfortunately this execution paradigm would be extremely limiting both in terms of the amount of concurrency and the granularity of access, that is, the locking level. Although such an approach is valid, the object model is significantly more powerful and expressive so its nested structure should be fully exploited. Therefore we adapt the concepts from nested transactions to form the definition of object transactions. The approach is to carefully define nested transactions and then to apply these to objects. This chapter does not provide complete models for nested transactions but introduces relevant concepts as appropriate.

3.1.1 Nested Transactions

Relevant aspects of nested transactions are briefly introduced here. We assume an understanding of concepts such as descendants, parents, and transaction families rooted by a top-level transaction. Nested transactions motivate the subsequent discussion of transactions in object bases. The nomenclature defined earlier for operations and transaction termination for flat transactions is carried through.

Nested transactions are inclusive in that they contain their descendants in addition to the invoking transaction. A top-level transaction is a nested transac-

tion that has no ancestors and we call it and its descendants a transaction family. Transaction families must appear atomic to other transaction families. We adopt a uniform view of transactions whereby every transaction within a family, from the top-level transaction to the leaf transactions, are themselves nested transactions. The differences being that leaf transactions have no descendants and top-level transactions have no parents.

Recall the definition of flat transactions from Chapter 2. The definition of nested transactions requires some additional nomenclature. A nested transaction τ_i has a set of descendant transactions $T_i = \{t_{i1}, \ldots, t_{in}\}$, of which $\chi_i \subseteq T_i$ are direct descendants or *children* of τ_i directly invoked by τ_i. An additional operation introduced for the purposes of nesting is the *pre-commit* (*pc*). Thus, operation k of nested transaction i is $O_{ik} \in \{r, w, pc\}$. The pre-commit is similar to the prepare-to-commit found in two-phase commitment; a transaction enters a pre-commit state before termination and remains there until told to terminate by a coordinator (in this case the top-level transaction).[1] By extending the notations for flat transactions, the operation set for τ_i is $OS_i = \{\bigcup_k O_{ik}\} \bigcup T_i$. The extension means that OS_i may contain reads, writes, pre-commits, and child transactions. Nested transactions also require the definition of a boolean function *depends*. It takes two operations as inputs and returns "true" if there is a dependence relation due to the internal semantics of the nested transaction. The justification for the use of this function follows the definition of a nested transaction.

Definition 3.1.2 A nested transaction is a partial order $\tau_i = (\Omega_i, <_i)$, where

1. $\Omega_i = OS_i \bigcup \{N_i\}$,

[1] Although it is possible to accomplish everything provided by the pre-commit by using it implicitly outside of the formal model, it is extremely useful and will be required in the study of reliability in a distributed environment.

2. (a) for any two $O_{ip}, O_{iq} \in OS_i$, if $O_{ip} = r(x)$ and $O_{iq} = w(x)$, for any x, $O_{ip} <_i O_{iq}$ or $O_{iq} <_i O_{ip}$,

   (b) for any two $O_{ip}, O_{iq} \in OS_i$, if $O_{ip} = t_{ij}$ and $depends(O_{ip}, O_{iq})$ or $depends(O_{iq}, O_{ip})$, then $O_{iq} <_i O_{ip}$ or $O_{ip} <_i O_{iq}$, respectively,

3. if $O_{ip} = pc$, O_{ip} is unique and $\forall O_{iq} \in OS_i$, $p \neq q$, $O_{iq} <_i O_{ip}$, and

4. $\forall O_{ip} \in OS_i$, $O_{ip} <_i N_i$. ■

Point (1) is obvious. Point (2) addresses the ordering relation of a nested transaction. Point (2a) states that conflicting read and write operations of this transaction must be ordered in some way. Point (2b) and the *depends* function are important and powerful constructs. Nested transactions that can invoke other nested transactions introduce several new semantic issues. We argue about the rationale of point (2b) with an example. If a nested transaction is defined without considering the internal semantics of the transaction, such as their parameters and return values, a total ordering of the invocations must be defined. This is because without knowledge of the semantics there is no way of defining the internal structure of the nested transaction and one must assume a serial ordering as defined through its implementation. For example, a nested transaction with only invocation operations $T_i : O_{i1}, O_{i2}, O_{i3}, \ldots, O_{in}$ would have an assumed totally ordered execution sequence of $O_{i1} < O_{i2} < O_{i3} < \cdots < O_{in}$. Unfortunately, this eliminates any possible concurrency inherent in the sub-transactions of the nested transactions. Further, the power of the nested transaction model is in the potential to parallelize the sub-transactions executing on behalf of a parent. Thus, Point (2b) allows for the interpretation of the semantics of the nested transaction to produce a partial rather than total ordering of the operations.

Several problems immediately present themselves but we will limit the discussion to two. (1) How is serialization to be accomplished and what is a suitable correctness criterion? (2) What techniques must be developed to determine which sub-transactions can execute concurrently and which must be delayed until others have completed? The first problem is extremely complex and proposals for a subset of this general model appear in the literature (see for example Beeri *et al.* [4]). The second problem is also extremely interesting and there has been some work in the area of dependency analysis aimed at parallelizing programs. For the purpose of this thesis, we assume that the *depends* relationship of Point (2b) has been defined *á priori* and any scheduler must be able to incorporate this information.[2]

Point (3) states that there is only one pre-commit per nested transaction and that all other operations of the nested transaction must occur before it. Point (4) ensures that all operations of a nested transaction occur before the transaction's termination.

This general definition for nested transactions does not address issues related to open versus closed executions of nested sub-transactions. Most of the literature describing open and closed execution models focus on determining the best technique to achieve a certain type of correctness based on some criterion. Whether or not a particular execution achieves this correctness criterion can be determined by inspecting its corresponding history. Although it is possible to explicitly define the transactions themselves as exhibiting open or closed characteristics it would be unduly restrictive. Therefore, orthogonal issues of open versus closed are deferred until the discussion of an execution model.

[2]Some early work proved very interesting but is beyond the scope of this thesis. Preliminary results are available in Graham *et al.* [12].

3.1.2 Transactions and Objects

Recall that object methods can invoke other methods so a nesting structure is useful to reason about method invocations. We assume that each "nested method call" is a nested transaction and apply the techniques described above. Users of an object base submit transactions that invoke a set of methods. A transaction submitted by a user must be atomic so the underlying system must ensure that the nesting of methods resulting from it is also atomic. This can be achieved through the adaptation of Definition 3.1.2 to object transactions.

From the user's view there is a set of objects which have methods that are invoked by transactions. Subsequent object transaction invocations are hidden from the user. Therefore, we have a level of abstraction between the object base and the user, in that the user need not be aware of *what* the object–based system does to perform requested actions as long as it guarantees a correct execution. This implies a two level architecture using two types of transactions, one for the user view and another for the system view. User transactions are composed of the invocation of methods on objects. The abstraction creates a flat view for users with the fundamental operations being the methods. Object transactions are a lower level of abstraction and are defined for the system view. These are essentially nested transactions on objects. The pragmatic reasons for the dichotomy are best explained by a formal model that makes allowances for both levels.

Object Transactions

We first define transactions for the system's view, called *object transactions*, which are the methods invoked on objects. The adaptation of nested transactions to object transactions requires the following additional nomenclature. The system contains a

set of objects $\Theta = \{o^1, \ldots, o^m\}$. x^e unambiguously denotes that data item x belongs to o^e. OT_i^e is object transaction i on object e. Recall from nested transactions that $T_i = \{t_{i1}, \ldots, t_{in}\}$ for nested transaction i. This is extended for object transactions such that $t_j^f \in T_i$ denotes a descendant of OT_i^e where the descendant transaction j executes on o^f. We also note that all read and write operations of an object transaction are on data items of the object where the transaction is executing. The definition of an object transaction is as follows:

Definition 3.1.3 An object transaction is a partial order $OT_i^e = (\Omega_i^e, <_i^e)$, where

1. $\Omega_i^e = OS_i \bigcup \{N_i\}$,
2. (a) for any two $O_{ip}, O_{iq} \in OS_i$, if $O_{ip} = r(x^e)$ and $O_{iq} = w(x^e)$, for any x^e, $O_{ip} <_i^e O_{iq}$ or $O_{iq} <_i^e O_{ip}$,

   (b) for any two $O_{ip}, O_{iq} \in OS_i$, if $O_{ip} = t_j^f$ and $depends(O_{ip}, O_{iq})$ or $depends(O_{iq}, O_{ip})$, then $O_{iq} <_i^e O_{ip}$ or $O_{ip} <_i^e O_{iq}$, respectively,
3. if $O_{ip} = pc$, O_{ip} is unique and $\forall O_{iq} \in OS_i$, $p \neq q$, $O_{iq} <_i^e O_{ip}$, and
4. $\forall O_{ip} \in OS_i$, $O_{ip} <_i^e N_i$. ■

The difference between nested and object transactions is that a nested transaction can access all of the data items of a database while an object transaction can only access the subset of data items at a particular object. The additions made to the definition reflect this by making the affected objects and corresponding data items identifiable.

User Transactions

The user's view requires the definition of a special transaction called the *user transaction*. This is what the user of an object base would write and is based on how the user views the object base. A user transaction invokes object transactions and can be viewed as the top-level transaction of nested object transaction families. This definition uses the nomenclature of nested and object transactions, as given above. The following additions, particular to user transactions, are made to the nomenclature. UT_i identifies user transaction i. $O_{ip} = t_j^e$ denotes operation p of a UT_i, which invokes object transaction j on o^e. Note that since there are no read, write, or pre-commit operations for user transactions, we know that all O_{ip} are transactions. The set of operations for user transaction i is $OS_i = \bigcup_k O_{ik}$. The definition of a user transaction is as follows:

Definition 3.1.4 A user transaction is a partial order $UT_i = (\Sigma_i, \prec_i)$, where

1. $\Sigma_i = OS_i \bigcup \{N_i\}$,
2. for any two $O_{ip}, O_{iq} \in OS_i$, if $depends(O_{ip}, O_{iq})$ or $depends(O_{iq}, O_{ip})$, then $O_{iq} \prec_i O_{ip}$ or $O_{ip} \prec_i O_{iq}$, respectively,
3. $\forall O_{ip} \in OS_i$, $O_{ip} = t_j^e$, $N_j = N_i$, and
4. $\forall O_{ip} \in OS_i$, $O_{ip} \prec_i N_i$. ■

Point (1) is obvious. Point (2) defines the ordering relation of a user transaction. It states that two operations of a user transaction that have a dependency must be ordered in the user transaction's ordering relation, as was previously explained for nested transactions. Point (3) ensures that all object transactions invoked by a user transaction come to the same termination decision as the user transaction.

Point (4) states that all operations of the user transaction must occur before the user transaction's termination.

3.2 Histories

Execution histories must be defined to discuss a correctness criterion for transactions on objects. The execution of object transactions on objects is described with an object history. These give the sequence of operations executed at a particular object, be they reads and writes on local data or the invocation of another method. OH^e identifies the *object history* for object e. The definition of an object history is as follows:

Definition 3.2.1 An object history is a partial order $OH^e = (\Omega^e, <^e)$, where

1. $\Omega^e = \bigcup_j \Omega^e_j$, where Ω^e_j is the domain of OT^e_j,
2. $<^e \supseteq \bigcup_j <^e_j$ where $<^e_j$ is the ordering relation of OT^e_j, and
3. for any two conflicting operations $p, q \in \Omega^e$, either $p <^e q$ or $q <^e p$. ■

Point (1) states that the set of operations of an object history is composed of all of the operations of the object transactions that have been executed on that object. Point (2) defines the ordering relation of an object history to contain the orderings of the object transactions that executed on the object. Point (3) states that the ordering relation of an object history will also contain an ordering for any two operations that conflict.

The interaction of objects through method invocations raises the need for a global history. This history records the invocation of all methods in the object

base, both from user and object transactions, along with the local operations of the object transactions. Defining such a global history requires the notion of a history with respect to user transactions. From this a global history can be defined as the combination of the object histories and the history of the user transactions. First, we define a *user transaction history* (UTH) as follows:

Definition 3.2.2 A user transaction history is a partial order $UTH = (\Sigma_{UTH}, \prec_{UTH})$, where

1. $\Sigma_{UTH} = \bigcup_i \Sigma_i$, where Σ_i is the domain of UT_i, and
2. $\prec_{UTH} \supseteq \bigcup_i \prec_i$, where $\prec_i$ is the ordering relation of UT_i. ■

Point (1) states that the operations of all of the user transactions in the system make up the set of operations in the user transaction history. Point (2) defines the ordering relation of the user transaction history to contain the orderings of the user transactions.

Definitions 3.2.1 and 3.2.2 provide histories for the two types of transactions in an object–based system. These histories can be combined into a global history to assess correctness. The user transaction history by itself is not sufficient because it does not capture the ordering of object transactions present in the object histories. The user transaction history is necessary because the global history can be considered correct if it is correct with respect to the user transactions. Correctness is then determinable by examining the execution orderings detailed in the object histories. Thus, the union of the object histories along with the execution sequence provided by the user transaction history yields the information required for serialization (see Section 3.3).

As every object transaction operation descends from some operation of a user transaction, the orderings of object transaction operations must be reflected in the ordering of user transaction operations. Thus, a *global object history* (GOH) is defined to contain all of the operations in the object base, enabling the object history orderings to be reflected in the ordering of user transaction operations.

Definition 3.2.3 A global object history is a partial order $GOH = (\Sigma_{GOH}, \prec_{GOH})$, where

1. $\Sigma_{GOH} = \Sigma_{UTH} \bigcup \{\bigcup_e \Omega^e\}$, and
2. $\prec_{GOH} \supseteq \prec_{UTH} \bigcup \{\bigcup_e <^e\}$, and
3. if $\exists O^e_{kp}, O^e_{lq} \in \Sigma_{GOH}$, such that $O^e_{kp} \prec_{GOH} O^e_{lq}$ and O^e_{kp}, O^e_{lq} descend from $O_{ik}, O_{jl} \in \Sigma_{UTH}$, respectively, then $O_{ik} \prec_{GOH} O_{jl}$. ■

Point (1) states that the set of operations of the global object history is composed of all of the operations of the user transaction history and the object histories. Point (2) defines the ordering relation of the global object history to contain the orderings of both the user transaction history and the object histories. Note that all possible orderings between conflicting operations are captured in the definitions for object histories and user transaction histories. Therefore, orderings regarding conflicting operations need not be defined within the global object history. Point (3) states that if there is an ordering between two object transaction operations, the user transaction operations that (indirectly) invoked them are identically ordered.[3]

[3]This shows a direct relationship between user transactions and object transactions but this can be extended to object transactions not directly invoked by a user transaction.

3.3 Serialization

Ultimately concurrency control algorithms are required but it is first necessary to define precisely what constitutes a correct execution. To do this, it is first necessary to define serial histories. A history that permits some amount of operation (or transaction) interleaving is considered correct if it is equivalent to a serial history, and is thus *serializable*.

For the purposes of this model, a serial history must be defined with respect to each of the histories defined above. This allows the most flexibility in designing concurrency control algorithms and facilitates a "bottom-up" approach when using the model. From these definitions we gain all of the information required to show correctness. The definition of an object-serial history is as follows:

Definition 3.3.1 An object history OH^e is object-serial (O-serial) iff, $\forall O_{ip}, O_{jq} \in \Omega^e$, if $O_{ip} <^e O_{jq}$, then $\forall O_{ik}, O_{jl} \in \Omega^e$, $O_{ik} <^e O_{jl}$. ■

This definition states that if there is an ordering between the operations of two object transactions, then all of the operations of one object transaction must precede any operation of another. This ensures that the object transactions will execute serially in that all of the operations of one will occur before all the operations of another at a specific object.

The definition of equivalence is as follows:

Definition 3.3.2 Two object histories at an object are *equivalent* iff they are over the same set of object transactions and they order conflicting operations in the same way. ■

Serializable object histories are now defined as follows:

Definition 3.3.3 An object history is *object-serializable* (O-serializable) iff it is equivalent to some O-serial object history. ■

This means we can reason about the "internal" correctness of each object in the object base individually. The inter-object correctness, however, remains undefined. Inter-object correctness is attained when the orderings within each object are compatible with one another, with respect to the user transactions. The orderings within each object are abstracted into orderings of object transactions, so that these orderings may be used to identify the correctness of the orderings of the user transactions.

The global object history contains the orderings contained in both the object and the user transaction histories. The definition of a correctness criterion for the global object history results in inter-object correctness through the serializability of the user transactions.

Global object histories are defined to be serial with the following definition.

Definition 3.3.4 A global object history is serial iff,

1. $\forall OH^e$, $e = \{1, \ldots, n\}$, OH^e is O-serializable, and
2. $\forall O_{ip}, O_{jq} \in \Sigma_{UTH}$, if $O_{ip} \prec_{GOH} O_{jq}$,
   then $\forall O_i, O_j \in \Sigma_{UTH}$, $O_i \prec_{GOH} O_j$. ■

This definition states that a global object history is serial if and only if all of the object histories are O-serializable and when there is an ordering of two operations from different user transactions, then all of the operations from the two user transactions must have the same ordering. This yields serial user transactions but also

gives the flexibility of only requiring serializability within each object, thereby attaining greater concurrency while maintaining serial execution at the desired level.

It is necessary to define the equivalence of two global object histories in order to define their serializability. The definitions for equivalence and serializability with respect to global object histories as follows:

Definition 3.3.5 Two global object histories are *equivalent* iff they are over the same set of transactions and they order conflicting operations in the same way. ■

Definition 3.3.6 A global object history is *serializable* iff it is equivalent to a serial global object history. ■

These definitions yield a correctness criterion upon which concurrency control algorithms can be judged. For the users of an object base to have a consistent view of the data they are accessing, the execution of the user transactions must be serializable. The definition of a serializable global object history enables this by ensuring an equivalent to serial execution ordering of the user transactions. This provides correct concurrent execution within the object base and the users will not see inconsistent data.

3.4 The Serializability Theorem

The global object history provides a means of linking user transactions to the object transactions they invoke. In practice, however, an implementation would probably not create a global object history. The history is a theoretical tool that demonstrates how the object and user transactions should relate by providing a means of

proving concurrency control algorithms correct. This requires gathering all needed information into one history and transforming it into an equivalent *serialization graph* that can be tested for acyclicity (see Bernstein *et al.* [5]). This section provides such a graph and theorem for the model.

The definition of a *global object graph* is as follows:

Definition 3.4.1 A global object graph (GOG) is a directed graph of a global object history (GOH): $GOG(GOH) = (\Upsilon, \Psi, \upsilon, \psi)$ such that,

1. Υ is a set of labeled vertices representing the user transactions,
2. Ψ is a set of labeled vertices representing the object transactions,
3. υ is a set of arcs, each connecting two vertices in Υ. Given two user transactions $UT_i, UT_j \in \Upsilon$, an υ–arc is added to $GOG(GOH)$ ($UT_i \rightarrow UT_j$) if an operation of UT_i precedes an operation of UT_j in the ordering relation of GOH, and
4. ψ is a set of arcs, each connecting two vertices in Ψ. A ψ–arc is added to $GOG(GOH)$ when given two object transactions on the same object (OT_i^e and OT_j^e) an operation of OT_i^e precedes an operation of OT_j^e in the ordering relation of GOH. The ψ–arc is added from the vertex for OT_i^e to the vertex for OT_j^e (ie. $OT_i^e \rightarrow OT_j^e$). ■

To illustrate the construction of a GOG, consider the following subset of an object history composed of three object transactions:

$$OH^e : r_1(x^e)w_2(x^e)w_3(y^e)r_2(y^e).$$

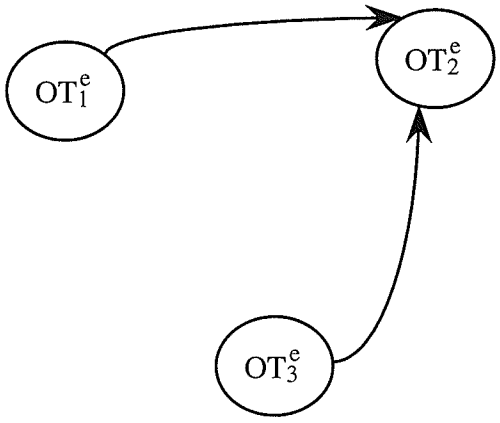

Figure 3.1: Example Graph made up of Ψ vertices and ψ–arcs.

Figure 3.1 depicts the pertinent sub-graph of this object history constructed according to the rules in Definition 3.4.1. Note that there is an edge between OT_1^e and OT_2^e due to the read-write conflict on x^e and an edge between OT_2^e and OT_3^e due to the read-write conflict on y^e.

Before we prove the utility and correctness of the GOG in defining precisely when a global object history is serializable, it is useful to provide some insight into why such a complex tool is required. This is accomplished by first showing why serialization of object transactions, by assuming the acyclicity of ψ–arcs, is insufficient for user transactions. Secondly, we argue that the acyclicity of υ–arcs is not sufficient by assuming they are and then indicate possible erroneous states. These examples will also provide intuition and motivation for the serializability theorem that follows.

Figure 3.2 demonstrates the need for a global characterization of the relationships between the orderings at different objects. This figure shows arcs between object transactions within their respective objects. Assume that OT_1^e and OT_1^f are operations O_{ip} and O_{iq} of UT_i, respectively, and OT_2^e and OT_2^f are likewise

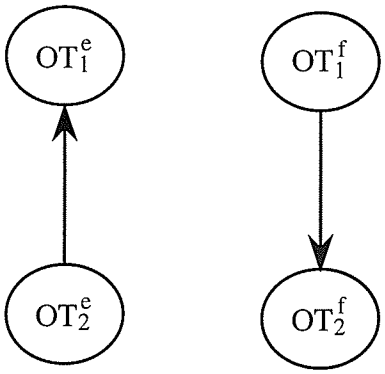

Figure 3.2: Example GOG with two objects and no Υ vertices.

operations O_{jr} and O_{js} of UT_j, since OT_2^e is serialized before OT_1^e at o^e and OT_1^f is serialized before OT_2^f at o^f (as shown in Figure 3.2), no cycles occur in the ψ–arcs and O-serializability is maintained at each object. However, when considering these object transactions as operations of the two user transactions, the orderings of $O_{jr} \prec_{UTH} O_{ip}$ and $O_{iq} \prec_{UTH} O_{js}$ arise, implying a cycle between UT_i and UT_j. Thus, serializing solely at the object level is insufficient to ensure inter-object correctness.

Alternatively, only user transactions could be considered. Using the same two user transactions defined above, consider the correctness ramifications if there is only serialization of user transactions. If UT_i were serialized before UT_j (as shown in Figure 3.3), then their respective object transactions should execute in the same ordering at each of the two objects. Consider o^e with the following possible object transactions at:

$$OT_1^e : r_1(x^e) w_1(x^e) pc_1 c_1,$$

and

$$OT_2^e : r_2(x^e) w_2(x^e) pc_2 c_2.$$

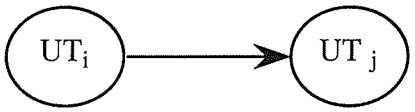

Figure 3.3: Example GOG with only Υ vertices.

Since there is no serialization performed at the objects, a possible object history is:

$$OH^e : r_1(x^e)r_2(x^e)w_1(x^e)pc_1c_1w_2(x^e)pc_2c_2.$$

This history is obviously not conflict serializable and causes the update made by OT_1^e to be lost. Therefore, although the user transactions are serialized, non-serializable executions are still allowed at the objects thereby causing the object base to enter an inconsistent state.

Finally, we argue that there exists a need for ψ–arcs even if υ–arcs are acyclic. Although correct executions could be achieved by only serializing the user transactions, if that serialization were enforced at the object level, another anomaly is possible if the invocation of the two object transactions OT_1^e and OT_1^e is from a single user transaction. Obviously, there are no υ–arcs because there is only one Υ–node, but it is impossible to guarantee the user transaction's operations within a particular object do not conflict without the ψ–arcs. Therefore, all user transactions would be correctly serialized with respect to each other so there are no cycles in the graph using υ–arcs, but it would not be possible to guarantee that two object transactions from the same user transaction would correctly be serialized at a particular object. These examples show the need to serialize both levels of transactions to ensure the global correctness of the object base. We have now illustrated why the global object graph of Definition 3.4.1 is necessary but it is still necessary to demonstrate its sufficiency.

Observe that the global object graph will actually be a set of disjoint sub-graphs corresponding to each of the object histories and the user transaction history. Note also that the global object graph will only be acyclic if all of the sub-graphs are acyclic. This graph leads to the following theorem.

Theorem 3.4.1 A global object history is serializable iff its global object graph is acyclic.[4]

Proof (if) Suppose that the global object graph is acyclic. Let the global object history be over $\Sigma_{GOH} = \{OT_1, \ldots, OT_n, UT_1, \ldots, UT_m\}$. ψ–arcs connect vertices of object transactions that are at the same object and υ–arcs connect vertices of user transactions. Since the GOG can be considered as a set of disjoint sub-graphs representing the individual objects and the set of user transactions, it is possible to consider each of the sub-graphs in isolation of the others when discussing their acyclicity (more formally, if at a given vertex in the GOG, it is impossible to traverse the edges of the graph in such a way that the traversal would encounter both ψ and υ–arcs and furthermore, if the traversal encounters a ψ–arc, the traversal will only encounter the ψ–arcs for one object). Therefore, without loss of generality assume the set of transactions only involve committed transactions and define the set as $\Sigma_{GOH} = \{T_1, \ldots, T_n\}$. Since all sub-graphs are disjoint, consider any two transactions $T_i, T_j \in \Sigma_{GOH}$ which will either be two user transactions or two object transactions on the same object. Since the graph is acyclic, each of the sub-graphs may be topologically sorted. Let $j_1, \ldots, j_m$ be a permutation of $1, \ldots, m$ such that $T_{j_1}, \ldots, T_{j_m}$ is the combination of the topological sorts of the sub-graphs. We must now show that global object history for these trans-

[4]The proof technique is adapted from Bernstein *et al.* [5, page 33].

actions is equivalent to a serial global object history. Let p, q be operations of T_i, T_j respectively. Suppose they conflict and $p \prec_{GOH} q$. By definition, there is an edge from T_i to T_j in the global object graph. Therefore, in any topological sort of the sub-graph containing T_i and T_j, T_i must appear before T_j. This implies that all operations of T_i must appear before any operation of T_j in a serial global object history containing the two transactions. Therefore, all conflicting operations will be ordered in the same way in both the global object history and a serial global object history over the same set of transactions. Therefore the global object history is equivalent to a serial global object history and is therefore serializable.

(only if) Suppose the global object history is serializable. Then there is a serial global object history equivalent to it. Consider any two transactions $T_i, T_j \in \Sigma_{GOH}$ which are either two object transactions on the same object or two user transactions, since when the global object history is serializable, all of the object histories are O-serializable. Assume an edge from T_i to T_j in the global object graph. This implies that the transactions have conflicting operations such that if the operations are p and q for T_i and T_j, respectively, then $p \prec_{GOH} q$. Since the global object history is equivalent to a serial global object history, this ordering also appears in the serial global object history. This implies that all of the operations of T_i appear before any operation of T_j in the serial global object history. Suppose that the global object graph has a cycle. Without loss of generality assume that the graph has a cycle with edges from T_1 to T_2, T_2 to T_3, ..., and T_k to T_1. This means that in the serial global object history, all the operations of T_1 appear before T_2, all the operations of T_2 appear before T_3, and that all operations of T_k appear before T_1. This means that all operations T_1 appear before T_1. Clearly absurd, so

the global object graph must by acyclic. ■

This theorem shows that algorithms conforming to this model will only produce serializable global object histories if they ensure that the global object graph is always acyclic. The first part of the theorem shows that if the global object graph is acyclic then the global object history is equivalent to a serial global object history that is a topological sort of the graph. This also shows that the global object history can be equivalent to more than one serial global object history since a graph can have more than one topological sort. The second part of the theorem shows that if the global object history is serializable then the global object graph will be acyclic. This is because the history will be equivalent to a serial history that is a topological sort of the global object graph.

The use of a variety of node and arc types causes the above theorem to be more complex than traditional serializability theorems. It was argued earlier that the complexity is required to ensure proper serialization of transactions. The first example illustrated that the use of only ψ–arcs resulted in an inconsistent global view. The second example illustrated how υ–arcs alone results in inconsistent object transaction executions unless severe restrictions on object access were included. (It turns out that the only concurrency achievable will be at the object level [34].) The final example demonstrated that it is possible for a single user transaction to execute on a single object non-serializably. Unfortunately this could lead to inconsistent execution of the atomic user transaction operations. Therefore, Theorem 3.4.1 is both necessary and sufficient.

Chapter 4

Concurrency Control

The model presented in Chapter 3 allows for the modular design of concurrency control algorithms for both user and object transactions. A concurrency control algorithm can be designed to manage both of these types of transactions independently. The model does not enforce autonomy between the control of object and user transactions, thus allowing for the sharing of information that can be used to aid the concurrency control mechanisms. This chapter defines separate algorithms to manage the user and object transactions.

Recall the architectural framework presented in Chapter 1. This architecture defines the relations between all software components of transaction management in an object–based system. The purpose of this chapter is to define how the Method Scheduler and Object Schedulers should be developed. Modularity rises from the use of the Object Schedulers to independently handle the object transactions while the Method Scheduler ensures the serializability of the user transactions (the global serializability). The Method Scheduler uses information from the Object Schedulers to maintain the global correctness of the user transactions. The global object graph

defined in Chapter 3 allows for the discussion of the correctness of these algorithms by using the Ψ vertices and ψ arcs to show the correctness of the Object Schedulers and the Υ vertices and υ arcs to show the correctness of the Method Scheduler.

The remainder of this chapter proceeds as follows. Section 4.1 expands on the architecture introduced in Chapter 1 to describe the details of the components whose algorithms are introduced in this chapter. Examples illustrating the functionality of the algorithms required by the architecture are given in Section 4.2. Section 4.3 discusses the Object Manager's algorithm for supporting communication. Section 4.4 presents the Method Scheduler's algorithm. Section 4.5 presents the underlying concepts and an algorithm for Object Schedulers. The correctness of the algorithms is discussed in Section 4.6. The chapter concludes with a discussion of the algorithms in Section 4.7.

4.1 A Concurrency Control Architecture

Recall the architectural framework of Chapter 1. This section expands on that architecture by discussing the specific messages that must be passed between the components responsible for concurrency control. Figure 4.1 depicts an overview of the concurrency control architecture for object bases. It shows two components, an *Execution Monitor* and an *Object Processor*. The Execution Monitor receives user transactions from users, pre–processes them, and submits requests for object transaction execution to the Object Processor. The Object Processor manages the executions and returns the results to the Execution Monitor. The Object Processor accesses persistent storage to retrieve object data required for the executions.

Recall that the Execution Monitor is composed of a *Transaction Manager* and *Method Scheduler*. The Transaction Manager receives user transactions and

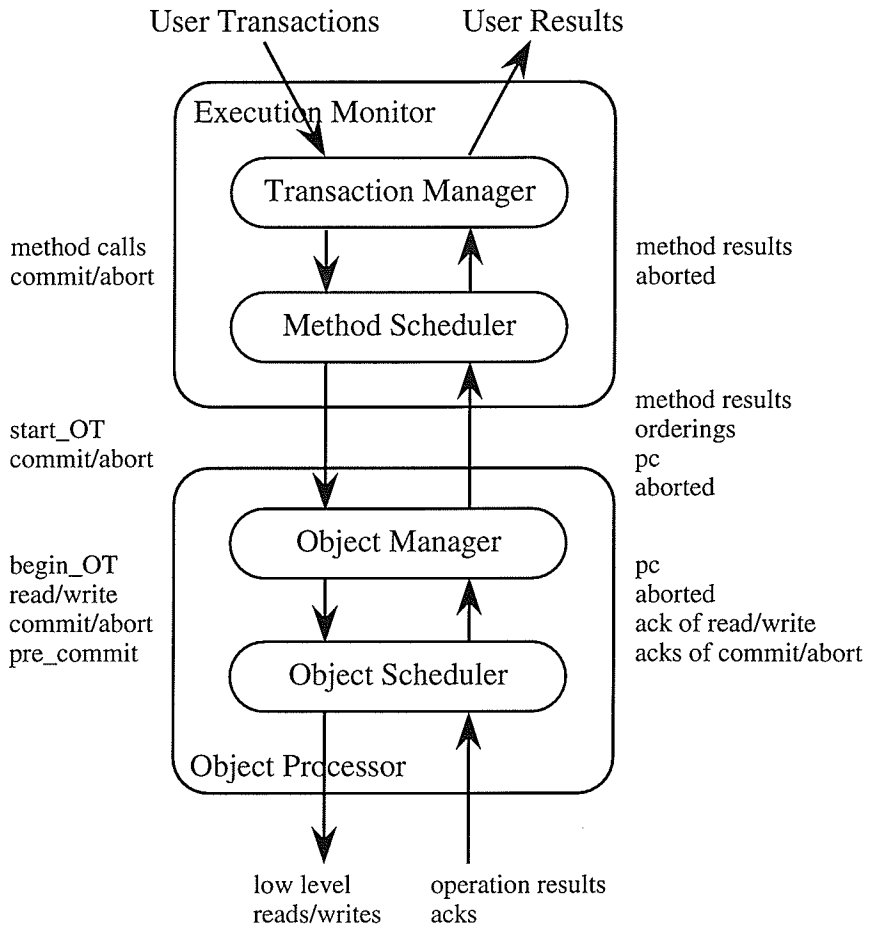

Figure 4.1: Overview of Concurrency Control Components

relays operations (requests for method execution) to the Method Scheduler. The operations are the pair O_{ip} and m_t^e, O_{ip} identifying user transaction i operation p and m_t^e identifying method t of object e. The Transaction Manager will submit a commit operation to the Method Scheduler following the submission of the methods of a user transaction. When the Method Scheduler receives either a pre_commit or aborted message from each of the object transactions submitted (as a $pc(O_{ip}.m_t^e)$ or $aborted(O_{ip}.m_t^e)$, respectively) it makes a final commitment decision and relays the final decision (possibly with results) to the Transaction Manager. The Method

Scheduler is responsible for scheduling user transaction operations serializably and passes the submission of an operation to the Object Processor for execution.

The Object Processor contains an *Object Manager* and one "logical" *Object Scheduler* per object. The Object Manager translates a method invocation into an object transaction suitable for submission to an Object Scheduler and facilitates communication between the Method Scheduler and the Object Schedulers. Therefore, its purpose is to relay messages between the two scheduler levels in a meaningful way. The Object Manager sends method invocations received from the Method Scheduler to the appropriate Object Schedulers as operations of object transactions. This requires the composition of each object transaction by attaching a transaction initiation command (begin_OT) to it and a request to prepare to commit the transaction at the end (pre_commit). Finally, with the submission of a user transaction's commit (abort), a commit (abort) operation is sent to each Object Scheduler accessed by the user transaction. This is accomplished by sending a $commmit(O_{ip}.m_t^e)$ $(abort(O_{ip}.m_t^e))$ to the Object Schedulers for each object transaction of a user transaction.

An Object Scheduler executes the data item read and write operations of the object transactions submitted to it. It must ensure conflict serializability at the data item level. We demonstrate that traditional mechanisms such as two-phase locking [5] with suitable modifications are appropriate (see Section 4.5). If we assume, for the moment, that the object scheduling algorithm is correct, this will continue until all read and write operations complete or the detection of a conflict.

Information must be relayed from the Object Scheduler to the Method Scheduler when any of these three significant events occur. First, if the pre-commit operation arrives the object transaction has "completed" and must wait for the final commitment decision from the Transaction Manager. Second, if the scheduler or the

object transaction itself aborts the object transaction, the Method Scheduler must be notified. Note that such an abort implies that the object transaction must relinquish any resources held at the time. In each case, a message is sent to the Method Scheduler indicating the user transaction operation ($O_{ip}.m_t^e$) that pre-committed or aborted. Thirdly, a nested method invocation may be required. Servicing this request is accomplished by passing it up to the Method Scheduler where it is bound to the appropriate user transaction and processed as any other method invocation. A message is sent which relates the method invocation to the user transaction operation that initiated the object transaction invoking the method. When a conflict is detected (as above) or one of these cases occur the Object Manager binds the event to a specific user transaction and relays the information to the Method Scheduler. Any ordering induced by the Object Scheduler (eg. conflicting operations) must be relayed to the Method Scheduler where the ordering is translated into an ordering of user transaction operations. The ordering is sent as a pair of user transaction operations with the first of the pair ordered before the second.

The Method Scheduler uses these orderings to serialize user transactions. Since the orderings are between the operations of user transactions they must be reflected in their ordering *per se*. A pre-commit indicates that an operation of a user transaction has completed. The Method Scheduler receives this information so that it can determine if and when to commit the user transaction. An abort implies the unsuccessful execution of an operation and results in the Method Scheduler immediately aborting the entire user transaction. This is extremely pessimistic but a less restrictive approach is beyond the scope of this thesis. Finally, when a user transaction completes the Transaction Manager is notified and the results (or failure) of the operations are returned to the user.

4.2 An Example Object Base

Before presenting the scheduling algorithms, a discussion of some scheduling problems that will be encountered is useful. A number of examples of the execution of user transactions are presented for the object base given in Figure 4.2. There are three objects each containing a set of attributes (S^1 for o^1) and methods (M^1 for o^1). A method is a sequence of read and write operations on the attributes of the object. Method invocations may also be present in a method, such as the invocation of m_1^3 of o^3 in m_3^1 of o^1. Each method ends with a pre-commit operation. The operations of each method are given in the order that they will be submitted for execution (based on the semantic definition of the method through some high-level language or query).

The independent scheduling of methods at each object introduces difficulties in the scheduling of the user transactions. The following examples demonstrate this for direct and indirect (via nested method invocation) serialization errors, respectively.

Example 4.2.1 Consider the following user transactions submitted to the object base:

$$UT_1 : m_1^1; m_1^2; c_1$$

$$UT_2 : m_2^1; m_1^2; c_2.$$

A possible user transaction history produced by their execution is:

$$UTH : O_{11}.m_1^1; O_{21}.m_2^1; O_{22}.m_1^2; O_{12}.m_1^2; c_2; c_1.$$

o^1 : S^1 : a^1, b^1, c^1

M^1 : $m_1^1 : r(a^1)r(c^1)w(b^1)pc_1^1$

$m_2^1 : r(b^1)w(a^1)pc_2^1$

$m_3^1 : r(a^1)m_1^3 w(c^1)pc_3^1$

o^2 : S^2 : x^2, y^2

M^2 : $m_1^2 : r(x^2)w(y^2)pc_1^2$

$m_2^2 : m_2^3 w(x^2)pc_2^2$

o^3 : S^3 : e^3, f^3, g^3

M^3 : $m_1^3 : r(g^3)w(e^3)pc_1^3$

$m_2^3 : r(f^3)w(g^3)pc_2^3$

$m_3^3 : r(e^3)w(f^3)w(g^3)pc_3^3$

Figure 4.2: An Object Base

Since the methods conflict and assuming that the Object Schedulers serialize the methods in the order shown above, $O_{11} \prec O_{21}$ and $O_{22} \prec O_{12}$ and the user transactions have been incorrectly scheduled. ■

This example illustrates the problem with optimistically submitting the operations of a user transaction for execution. The independence of the objects implies that incompatible orderings can result. There are two possible means of correcting this error. The error could be detected after the user transactions have completed, causing the abortion of one of the user transactions so that the history remains serializable. Alternatively, invocations of a user transaction can be blocked so that

incompatible orderings cannot be produced. A combination of these techniques is employed by the Method Scheduler introduced in Section 4.4. This example illustrates the problem with respect to direct method invocations but a more subtle scheduling difficulty can arise, as shown in the following example.

Example 4.2.2 Consider the following user transactions submitted to the object base:

$$UT_1 : m_3^1; m_1^2; c_1$$

$$UT_2 : m_2^2; c_2$$

$$UT_3 : m_3^3; c_3.$$

A user transaction history produced by their execution is:

$$UTH : O_{11}.m_3^1; O_{11}.m_1^3; O_{31}.m_3^3; O_{21}.m_2^2; O_{12}.m_1^2; O_{21}.m_2^3; c_1; c_3; c_2.$$

The orderings induced by the Object Schedulers implies that $O_{11}.m_1^3 \prec O_{31}.m_3^3$, $O_{21}.m_2^2 \prec O_{12}.m_1^2$, and $O_{31}.m_3^3 \prec O_{21}.m_2^3$. This results in an incorrect scheduling of the three user transactions. ■

This example introduces the complexities involved in the concurrency control of nested transactions. A serialization error occurs even though neither UT_1 nor UT_2 directly invoke methods that conflict with the method invoked by UT_3. This implies that all nesting must be considered in the scheduling decisions made with respect to user transactions. This example also introduces the association of nested invocations with a user transaction. The discussion of nesting requires that the invocations be related to the user transaction that is the top-level transaction.

Each nested invocation is linked to the operation of the user transaction that is at the root of the call tree (as done with the invocation of m_1^3 by m_3^1 by having both methods associated with O_{11}).

Another difficulty that must be handled by the algorithm is the commitment of a user transaction. The scheduling of a user transaction by independent components can result in internal serialization errors, as shown in the following.

Example 4.2.3 Consider the following user transaction submitted to the object base:

$$UT_1 : m_1^1; m_2^1; m_2^3; c_1.$$

Assume that the partial order of the user transaction indicates that $m_1^1 \prec_1 m_2^1$. First, if all operations have been submitted and the user transaction issues the commit operation before all of the methods have returned, the commit cannot be processed.

Once all methods have returned, the commit can proceed. A user transaction history for the user transaction's execution is:

$$UTH : O_{12}.m_2^1; O_{13}.m_2^3; O_{11}.m_1^1; c_1.$$

This history is incompatible with the partial order of the user transaction. A serialization error occurred within the user transaction and the commit should not be allowed to proceed. ■

The semantic complexity of a user transaction results in the need to analyze the execution of the transaction so that it does not contradict the semantic meaning of the transaction. This analysis and the commitment of the user transaction

must be delayed until all methods have completed. This form of commitment is analogous to the two-phase commit protocol [5]. The user transaction must wait for the termination decision of all of its descendants and, based on their decisions, makes a termination decision which is relayed to all of the descendants.

All of these problems must be addressed by the concurrency control algorithms of an object base. There are many subtle serialization errors that can occur due to the use of nesting and independent schedulers. The algorithms presented in the next three sections ensure correct serialization by eliminating the problems discussed above through avoidance and correction mechanisms.

4.3 Communication through the Object Manager

The Object Schedulers must interact with the Method Scheduler so a communication facility is required to link method invocations with object transactions. The Object Manager, as described in Section 4.1, is responsible for submitting the object transactions to their corresponding schedulers. Therefore, the Object Manager must record the information needed to identify the invoker of an object transaction and corresponding user transaction.[1] The Object Manager must also relay information from the Object Schedulers to the Method Scheduler. It is assumed that a reliable message passing system exists for inter-process communication. This implies that each procedure that implements the sending of information between the components of the system must employ a handshaking and blocking mechanism to ensure that all messages are received reliably. In addition, a message queuing

[1]This is necessary for nested method invocation.

mechanism, using *insert* and *front* operations, exists for adding a message to the end of a queue and removing a message from the front, respectively. This is a reasonable assumption because protocols for acknowledgment and message timeouts are well understood [23].

Figure 4.3 gives the communication protocol provided by the Object Manager. This algorithm uses an *event loop* that reacts to message traffic between the Method and Object Schedulers. Before discussing the details, a few procedures and data structures are required:

object_msgs : A queue of messages awaiting service at the Object Manager. These messages can come from the Method Scheduler or any of the Object Schedulers completing or communicating results of active transactions.

call_table : A table storing the invocation information required for each object transaction currently executing. The rows are object transactions identified by object identifiers and transaction numbers. There are two columns, the first identifying the user transaction operation and the second identifying the method invoked. The following table manipulation routines are required:

insert_row : Adds a new row to the table for an object transaction. Data required by the table is passed as parameters.

delete_row(id) : Removes the row from the table.

sendtoMS : Sends a particular message to the Method Scheduler by inserting the message at the back of the message queue for the Method Scheduler and by employing handshaking which causes blocking until an acknowledgment is received.

```
begin
  loop;
    case front(object_msgs) of

        ! Messages from the Method Scheduler

      begin_OT(O_ip,m_t^e):
        OT_j^e ← generate_unique_trans(m_t^e);
        insert_row(OT_j^e,O_ip,m_t^e);
        submit OT_j^e to o^e
      commit(O_ip.m_t^e):
        ∀OT_j^e, if call_table[OT_j^e,1].call_table[OT_j^e,2] = O_ip.m_t^e then
          submit commit(OT_j^e) to o^e;
          delete_row(OT_j^e)
      abort(O_ip.m_t^e):
        ∀OT_j^e, if call_table[OT_j^e] = O_ip.m_t^e then
          submit abort(OT_j^e) to o^e;
          delete_row(OT_j^e)

        ! Messages from the Object Schedulers

      pre_commit(OT_j^e):
        sendtoMS(pc(call_table[OT_j^e,1]))
      aborted(OT_j^e):
        sendtoMS(aborted(call_table[OT_j^e,1]));
        delete_row(OT_j^e)
      ordering(OT_i^e,OT_j^e):
        sendtoMS(ordering(call_table[OT_i^e,1], call_table[OT_j^e,1]))
      invoke(OT_j^e,m_t^f):
        sendtoMS(invoke(cal_table[OT_j^e,1],m_t^f))
      NULL:
        ;
    end case
  end loop forever;
end
```

Figure 4.3: Object Manager's Communication Protocol

generate_unique_trans : Translates the method invocation passed from the Method Scheduler to an object transaction that includes a begin_OT, the method invocation and a pre_commit operation. Additionally, an object transaction identifier specific to the object where the transaction is to execute is created. This is accomplished by using the object's identifier and a new unused transaction number for that object.

submit : Sends a command (eg. commit or abort) or an object transaction to an object for scheduling. For an object transaction, it sends the operations to the Object Scheduler in the order specified by the invoked method. It employs handshaking as is done with *sendtoMS*.

The event loop receives various messages from the Method and Object Schedulers. It executes as a daemon, waiting for messages and processing them as they arrive. The Object Manager reads the messages from its queue and processes them as follows. When the Method Scheduler sends a *start_OT* its arrival data is stored in *call_table*. First an object local identifier is generated. The table stores the unique identifier (OT_i^e), the user transaction operation, and the method_id (see Figure 4.3).

Ultimately, the set of object transactions submitted on behalf of a user transaction will need to commit or abort. The commitment decision must be passed from the Method Scheduler to the relevant Object Schedulers by transmitting a *commit* or *abort* message through the Object Manager. A *commit* message indicates that the relevant object transaction should be committed. An *abort* message is translated into abort operations at specific objects. An abort of an object transaction is then submitted to its Object Scheduler. For both messages, the rows of *call_table* for the object transaction that have terminated are deleted.

A *pre_commit* is a message sent by an Object Scheduler to notify the Method Scheduler of an object transaction's completion. Similarly, if an object transaction has aborted the *aborted* message must be relayed to the Method Scheduler. In both cases, the message sent to the Method Scheduler indicates the user transaction and corresponding operation that completed. The arrival of an *aborted* message permits the Object Manager to delete the relevant entry from the table. Execution order information is passed from the Object Schedulers to the Method Scheduler. These are relayed to the Method Scheduler with an *ordering* message. An *invoke* message from an Object Scheduler is associated with a user transaction operation by consulting the *call_table* for the specified object transaction. This user transaction operation is sent to the Method Scheduler, along with the method identifier. The nested method invocation is related to the user transaction so that it can be scheduled as a descendent.

We now present the concurrency control algorithms employed by the Method and Object Schedulers. First we describe the user level concurrency control and then present the object level.

4.4 User–Level Concurrency Control

The Method Scheduler accepts method invocations, from the Transaction Manager, on behalf of user transactions and sends them to the Object Manager for distribution to the appropriate Object Schedulers. The purpose of the Method Scheduler is to ensure that the user transactions executing on an object base do so serializably. This section presents a graph construction mechanism, called a $\mathcal{DAX}$ (for $\mathcal{D}$irected $\mathcal{A}$cyclic construction), where the vertices are user transactions and the arcs are ordering relations between the user transactions. The graph algorithms

of Resende and El Abbadi [26] construct a graph containing vertices and arcs for all transactions active in an object base while this algorithm only creates vertices and arcs for user transactions. We use information from the Object Schedulers to infer ordering information about the user transactions. This permits the use of separate algorithms for local object scheduling and global (inter-object) scheduling. Hadzilacos and Hadzilacos [13] argue that such an approach is the most appropriate concurrency control technique for object bases.

Recall that all operations of the user transactions and all operations submitted to the Method Scheduler are method invocations. The Method Scheduler uses a *hybrid* strategy in submitting method invocations. It optimistically submits methods when no ordering information for its user transaction is available but will pessimistically block methods when ordering information passed "up" from the Object Scheduler indicates invalid serializations at an object may result. Optimizations of the pessimistic approach are currently being investigated [12] but these discussions are beyond the scope of this thesis. The algorithm describes the construction and manipulation of the $\mathcal{DAX}$ and how the Method Scheduler submits transactions to the Object Manager.

The algorithm requires the following data structures and functions:

$\mathcal{DAX}$: A graph constructed by the Method Scheduler, where vertices are the active user transactions. The following routines manipulate the graph:

node(i) : Returns **true** if a node exists in $\mathcal{DAX}$ with identifier i.

create_node(i) : Adds a node to the graph with the identifier i.

delete_node(i) : Removes the specified node from the graph and any incident arcs.

add_arc(i, j) : Adds an arc from the node i to the node j.

delete_arc(i, j) : Removes the arc from i to j.

incoming(i) : Returns the set of nodes whose arcs are incident with i.

cycle : Returns **true** if there is a cycle in the graph, else **false**.

sendtoOM() : Sends a message to the Object Manager by inserting the message at the back of the message queue for the Object Manager and by employing handshaking which causes blocking until an acknowledgment is received.

passivate : Stops the given user transaction from executing further. It adds the user transaction identified to a set of all passive user transactions.

activate : Tests the set of passive user transactions for possible reactivation. Each user transaction node that has no incoming arcs is removed from the passivated set and is permitted to continue executing.

pause (UT_i) : A function that suspends further execution of an executing routine until restarted. This can be thought of as a thread that suspends itself so UT_i's execution path through the routine is halted (not the routine itself).

restart (UT_i) : Restarts a paused thread at the next executable line.

executing.UT_i : The set of operations that have been submitted for execution on behalf of UT_i. The set is initialized when a node is created, in $\mathcal{DAX}$, for the user transaction and is deleted when the node is removed.

The Method Scheduler must handle a variety of messages from the Transaction and Object Manager (see Figure 4.1). The messages from the Transaction Manager are received as input to the *Initial Scheduling* part of the Method Scheduler depicted in Figure 4.4. The operations can be one of three types, a commit

```
begin
  input O_ip : An operation for UT_i
       m_t^e : The method being invoked by the operation

  if not node(i) then                                              (1)
    create_node(i);                                                (2)
  case O_ip of
    commit:                                                        (3)
      if executing.UT_i ≠ ∅ then                                   (4)
        pause (UT_i)                                               (5)
      else
      begin
        for each O_ip.m_t^e ∈ UT_i do                              (6)
         if ∃ ordering(O_ip.m_t^e,O_iq.m_u^e) then                 (7)
           if O_ip ≺ O_iq ∈ UT_i then                              (8)
             sendtoOM(commit(O_ip.m_t^e))                          (9)
           else for each O_ip.m_t^e do                             (10)
             sendtoOM(abort(O_ip.m_t^e));                          (11)
           break;                                                  (12)
      end
      delete_node(i);                                              (13)
      activate                                                     (14)
    abort:                                                         (15)
      for each O_ip.m_t^e do                                       (16)
        sendtoOM(abort(O_ip.m_t^e));                               (17)
      delete_node(i);                                              (18)
      activate                                                     (19)
    otherwise:
      if incoming(i) = ∅ then                                      (20)
      begin
        sendtoOM(start_OT(O_ip,m_t^e));                            (21)
        executing.UT_i ← executing.UT_i ⋃ O_ip.m_t^e               (22)
      end else
        passivate UT_i                                             (23)
  end case
end
```

Figure 4.4: The Method Scheduler Part 1 – Initial Scheduling

(line 3),[2] an abort (line 15), or a method invocation (line 20). A *commit* implies that the user transaction specified in the operation is ready to make its updates persistent. If any operations have not pre-committed (line 4) then the user transaction is paused until the pending operations have completed. Otherwise, the user transaction can commit if no internal inconsistencies exist (line 6). An inconsistency arises when an ordering generated by an Object Scheduler contradicts an ordering implied by the partial order of the user transaction (defined in Chapter 3) (line 7–8). This results in the abortion of the user transaction and all of its associated object transactions (line 10–11). Following this, the node of the user transaction is removed from $\mathcal{DAX}$ (line 13) and **activate** is issued to test for any user transactions that can be reinitiated (line 14).

An *abort* implies that the user transaction wants all of its operations aborted. The object transactions for the user transaction are aborted by sending requests to the Object Manager (line 16–17). The node of the user transaction is removed from the graph (line 18) and the passivated transactions are tested for any that can be reinitiated (line 19). For the invocation of a method, the Method Scheduler first tests to see if the operation can proceed by testing the graph for any arcs coming into the node representing the user transaction (line 20). If there are, it implies that there is a user transaction serialized before it on some object. In order to maintain consistency across objects, the current (and any subsequent) operation of the user transaction should not be allowed to proceed because it could result in an ordering that is contrary to the ordering already represented in the $\mathcal{DAX}$. The submission of the operation could result in an incorrect ordering and the abortion of the operation. This is avoided by passivating the corresponding user transaction (line 23). If the operation can proceed, the Object Manager is sent a request to begin an object

[2]These line numbers reference those in the algorithm depicted in Figure 4.4.

transaction on behalf of the user transaction (line 21) and the operation identifier is appended to the set of executing operations for the user transaction (line 22).

Messages are also received from the Object Manager including: operations, transaction terminations, or ordering information. The algorithm for processing these messages is given in Figure 4.5.

```
begin
  input msg : a message sent from the Object Manager

  if msg = pc(O_ip.m_t^e) then                                    (1)
  begin
    executing.UT_i ←executing.UT_i − O_ip.m_t^e;                  (2)
    restart (UT_i)                                                (3)
  end
  if msg = aborted(O_ip.m_t^e) then                               (4)
    execute (abort(UT_i))                                         (5)
  if msg = ordering(O_ip.m_t^e,O_jq.m_u^e) then                   (6)
  begin
    add_arc(i,j);                                                 (7)
    if cycle then                                                 (8)
    begin
      delete_arc(i,j);                                            (9)
      sendtoOM(abort(O_ip.m_t^e));                                (10)
      execute (O_ip,m_t^e)                                        (11)
    end
  end
  if msg = invoke(O_ip,m_t^f) then                                (12)
    execute (O_ip,m_t^f)                                          (13)
end
```

Figure 4.5: The Method Scheduler Part 2 – Termination Scheduling

The algorithm uses a message passing routine called **execute** which sends the specified operation to the algorithm in Figure 4.4. This is needed to effectively emulate an abort message or a nested method invocation coming from the transaction manager. The *pc* message indicates that the specified operation has completed (line 1).[3] The Method Scheduler removes this operation from the set of executing operations (line 2) and issues a **restart** (line 3). This allows a user transaction waiting for a pre-commit to reinitiate and attempt to commit. *Aborted* (line 4) indicates that the given operation was unable to complete at an object and the user transaction that initiated the operation is forced to abort since it is impossible to complete all of its tasks. Therefore, we immediately abort any object transactions by issuing an abort of the user transaction (line 5).

The *ordering* message (line 6) indicates that an ordering was induced at an object. The operations specified in the message have been ordered such that the first has been serialized before the second. An ordering has been induced between two user transactions and an arc is added to $\mathcal{DAX}$ (line 7) from the node of the first operation's user transaction to the node of the second's user transaction. The graph is then tested for a cycle (line 8), which would imply a serialization error. If a cycle exists, it must have been induced by the arc that has just been added and the operation that caused the ordering by blocking the other operation is aborted and submitted for re-execution. The arc is removed from $\mathcal{DAX}$ since the ordering no longer exists (line 9–11). Finally, an *invoke* message means that a method invocation is being relayed to it from the Object Manager (line 12). This method is submitted for execution as a part of the user transaction operation that has as its descendent the object transaction that invoked the method (line 13).

[3]The following line numbers reference those in the algorithm depicted in Figure 4.5.

4.5 Object–Level Concurrency Control

This section introduces the modifications necessary to convert (strict) two-phase locking into object two-phase locking (O2PL). Other traditional concurrency control protocols could be modified in a similar manner but they are not discussed here. Based on the two–phase locking rules presented by Bernstein *et al.* [5, page 50]; the rules for the object two-phase scheduler are:

1. When the scheduler receives an operation $p_i(x^e)$ of OT_i^e, the scheduler tests if the lock for the operation conflicts with any lock held by another operation $q_j(x^e)$ of OT_j^e. If not, the lock is set and the operation is submitted for execution. If a conflict occurs the *call_table* is consulted to determine if the two object transactions belong to the same user transaction and if OT_j^e has pre-committed. If it has, the operation is submitted. Otherwise, the object transaction and its operation $p_i(x^e)$ is delayed. Finally, this implies an ordering between the two object transactions so the necessary $ordering(OT_j^e, OT_i^e)$ message is sent to the Object Manager.

2. Once a lock has been set for OT_i^e, the lock is held until the object transaction *commits* or *aborts*.

3. When an Object Scheduler receives a $commit(OT_i^e)$ ($abort(OT_i^e)$) from the Object Manager, it commits (aborts) the operations of OT_i^e and releases its locks.

4. When an Object Scheduler receives a pc_i from OT_i^e, it sends $pre\_commit(OT_i^e)$ to the Object Manager and waits.

5. When an Object Scheduler receives an a_i from OT_i^e, it sends $aborted(OT_i^e)$ to the Object Manager, aborts the object transaction's operations, and releases

its locks.

6. When an Object Scheduler receives an $invoke(m_t^f)$ from OT_j^e, it sends $invoke(OT_j^e, m_t^f)$ to the Object Manager.

Rule (1) prevents two object transaction from accessing object level data in a conflicting manner (often defined by a lock compatibility matrix). This is mandatory if the object transactions are from different user transactions, but special considerations are required if they are from the same user transaction. For example, if OT_j^e holds a lock needed by OT_i^e but OT_j^e is pre-committed it is impossible for it to get its lock because OT_j^e and OT_i^e must both pre-commit before either can commit. This is a *deadlock* that is "internal" to the user transaction. This anomaly is avoided if an object transaction is permitted to execute while holding locks conflicting with those of other object transactions from the same user transaction but only if the conflicting transactions have pre-committed. This guarantees that they have "completed" and will not execute more operations, such as a unilateral abort.

Rule (2) enforces strict two-phase locking in that no locks are released until the termination of an object transaction. Strictness is necessary to avoid the very difficult problem of *cascading aborts* [5]. The remaining rules provide the functions necessary for the object two-phase locking algorithm to interact with the Method Scheduler (via the Object Manager). Rule (3) defines the responses that are made to terminate messages received from the Object Manager. When the Object Manager receives a request to initiate a commit or abort, it notifies the object schedulers which perform the appropriate operations and release locks held by the specified object transaction.

Rule (4) defines the activity required when a pre-commit is issued. The Method Scheduler and Object Manager must be notified and the object transaction

is *passivated* since it will no longer issue any operations. Rule (5) is similar to Rule (4) except that it defines the activities necessary when an object transaction aborts. Once again the Method Scheduler and Object Manager must be notified but, in this situation the object transaction is aborted and its locks are released so blocked transactions can resume. Nested method invocation is handled by Rule (6). If an object transaction invokes a method as an operation, a request is made for its execution by sending the method and object transaction identifiers to the Object Manager.

4.6 Correctness

The schedulers discussed above must ensure serializability for them to be useful. The global serializability of an object base was defined in Chapter 3. A global object history is created based on user and object transactions and is serializable if the object histories are serializable and the user transaction history is equivalent to some serial history. A graph technique was introduced for showing that a global object history is serializable if and only if a graph constructed from the history is acyclic. This section discusses such a graph construction for the Method and Object Schedulers and argues that the graph is always acyclic. In addition, the techniques used to ensure correct nesting and atomic commitment are shown to be valid with respect to serializability and atomicity requirements.

Since a serializable global object history requires serializability at the objects, the Object Scheduler introduced in Section 4.5 is discussed first. Recall that the global object graph has two types of vertices and arcs. Ψ represents the set of vertices for object transactions and ψ represents the directed arcs between those vertices. Since a ψ–arc will only connect two object transactions from the same

object, we can restrict the discussion to one object without loss of generality.

The Object Scheduler employs strict two-phase locking, which is conflict serializable (as discussed in Chapter 2 and Bernstein *et al.* [5]). This implies that a sub-graph made up of Ψ vertices is acyclic if the scheduler employs strict two-phase locking. Unfortunately, the Object Scheduler adds additional functionality which must be taken into account. The reception of a commit or abort message from the Object Manager or the sending of pre_commit or abort messages to the Object Manager does not affect serializability, they ensure the atomicity of termination which will be discussed with respect to the Method Scheduler.

The execution of multiple object transactions by a single user transaction on an object introduces a deadlock anomaly as discussed with respect to Rule (1) of the Object Scheduler in Section 4.5. The anomaly is avoided by permitting transactions within a transaction family to hold conflicting locks. The transactions can only hold conflicting locks if all other transactions holding the locks have pre-committed. This is *lock migration* [21] whereby a transaction gives its lock to another transaction in its transaction family once it has completed execution. Only one active transaction within a transaction family may hold a conflicting lock. Therefore, any other transaction requiring a lock that conflicts with the lock is blocked until the blocking transaction pre-commits. Therefore, an ordering will result between transactions of the same transaction family (if they conflict) that results in an acyclic graph. Since a transaction does not let others within its transaction family obtain conflicting locks until it pre-commits, it can never be serialized before the conflicting transactions (since it has completed and will no longer obtain locks).

The invocation of a method by an object transaction does not affect the serialization of the object transaction unless it is on the same object. In this case,

it will be considered another transaction within the same transaction family and be handled using the lock migration technique discussed above. Other aspects of nested method invocation relate to the Method Scheduler and discussion of these issues is deferred until the discussion of the correctness of the Method Scheduler. Since strict two-phase locking and the additions made for objects result in acyclic graphs, the sub-graphs containing Ψ vertices and ψ arcs are acyclic. Therefore, all objects employing the Object Scheduler of Section 4.5 will produce O-serializable histories.

The design of the Method Scheduler assumes that correct Object Schedulers exist. If a correct Method Scheduler is employed with correct Object Schedulers, the global object history will be serializable. The sub-graph containing Υ vertices and υ arcs for the user transactions must be acyclic for this to be true. This graph can be constructed from the $\mathcal{DAX}$ by including an Υ vertex for each vertex added to $\mathcal{DAX}$ and subsequently adding the necessary arcs.

All arcs added to $\mathcal{DAX}$ become υ arcs which are removed from the serialization graph if they are removed from $\mathcal{DAX}$ due to a serialization error. If a user transaction aborts, the vertex and all incident arcs are removed from the serialization graph. The committed projection of the user transactions (the set of user transactions that have successfully terminated) can be extracted from the serialization graph and will always be acyclic. This is because the $\mathcal{DAX}$ is always acyclic. Since the serialization graph is constructed based on $\mathcal{DAX}$ and cycles are always removed from $\mathcal{DAX}$, the serialization graph with respect to committed user transactions will be acyclic. This holds true for both committed projections and after the completion of a set of user transactions (which is simply the committed projection containing all user transactions). Therefore, the global object graph is acyclic so global serializability is ensured.

Two unique aspects of the concurrency control algorithms discussed in this chapter are nested invocations and the commitment of a user transaction. Nested method invocations allow for complex computations within a user transaction to be hidden from the user. It also permits greater concurrency by allowing sub-transactions of a user transaction to execute concurrently. The technique employed here is to have the invocation of a method sent to the Method Scheduler. The invocation is related to a user transaction since it is logically an operation of that user transaction. It is a descendent of a particular operation of the user transaction in that the call path to the method invocation starts at a particular operation. Therefore, the method executes as a method invoked by that operation. This allows for orderings induced by the method to be linked to a particular user transaction operation. Thus, all orderings can be related to user transactions and all orderings induced at objects are reflected in the serialization of the user transactions. This technique is similar to that employed by multi-level transactions for proving the correctness of histories by flattening the call hierarchy [4]. Transactions at lower levels are considered operations at higher levels and if the lower levels are correct, they can be "flattened" into atomic operations.

The commitment of a user transaction employs an implicit two-phase commit protocol [5]. The initial phase consists of the user transaction submitting object transactions for execution and the object transactions returning pre-commits or aborts. The second phase consists of the user transaction making a termination decision and relaying the decision to the object transactions. The object transactions then implement the decision at their objects. The termination decision made by the user transaction is more complex than that of traditional two-phase commitment. In addition to committing if all object transactions reply with a pre-commit and aborting if one object transaction replies with an abort, the internal correctness

of the user transaction must be maintained. Every user transaction has a partial order due to its semantic definition [35]. The orderings must retain the partial order or the user transaction will have an internal inconsistency. Abortion of the transaction occurs upon the detection of an inconsistency.

4.7 Summary

All of the techniques employed by the concurrency control algorithms combine to provide executions that are atomic, nested, and serializable. The algorithm is correct with respect to the model defined in Chapter 3. Two-phase locking has been adapted to the object–based environment and the nesting of methods and lock migration have been introduced. This chapter also introduces techniques for the interchange of information between the two schedulers. Many optimizations to the algorithms are possible but are beyond the scope of this thesis. The algorithms have been introduced theoretically so that the important concepts can be discussed without dealing with the complexities of implementation.

Chapter 5

Conclusion

Concurrency control in object–based systems is a complex problem. This thesis examines the problem and contributes a set of algorithms for concurrency control. A model of transactions in an object–based system is introduced and a correctness criterion was defined for the transactions. This model is the basis of an architecture defining the software components of transaction management and in particular concurrency control. The architecture provides the framework from which the algorithms are defined.

Unlike previous correctness criteria, this thesis bases correctness on operation conflict and the syntax of methods. In addition, the model defines two types of transactions, one at the user level and the other at the object level. This allows for the natural dichotomy between what the user has access to (the objects' external specifications) and the implementation of objects through methods that access the internal data of the objects. The correctness criterion makes use of the two types of transactions to define serializability. Global serializability assumes that object executions are serializable. This is more general than homogeneously defining trans-

actions and defining global serializability as the equivalence to a serial execution of all transactions. Instead, more concurrency is allowed by only requiring *serializable* executions at the objects.

The algorithms introduced in this thesis provide for correct executions of transactions in object–based systems. A modified two-phase locking algorithm that makes allowances for nesting and communications with other schedulers was presented. In general, it shows the techniques required to transform traditional concurrency control algorithms so that they can be used in object–based systems. The user level algorithm maintains global serializability through a new technique that employs the construction of a graph during the execution of user transactions. The algorithm builds the graph using ordering information obtained from the objects and detects serialization errors as cycles in the graph. This thesis also introduced a protocol by which Object Schedulers communicate with the Method Scheduler. The protocol has to convert messages into forms that are readable by the schedulers as well as transform method invocations into transactions for submission to the appropriate Object Schedulers.

With respect to concurrency control only *first order* objects are of importance. That is, only objects that have been created and are active within an object–based system require concurrency control. Therefore, many aspects of object–orientation can be ignored in the design of concurrency control algorithms. Some of these object–oriented features, however, deal not only with the creation of objects but with how they will be used once they exist. It is important that concurrency control not hinder such features. One such feature is *composite* or *complex* objects defined through a *part-of* hierarchy [3]. Complex objects allow for objects to have other objects as a part of their definition. Therefore, methods invoked on an object may cause invocations on the objects that are a part of that object. The model and

algorithms presented in this thesis provide for this functionality through the use of nested method invocations and require no modifications in design.

Many aspects of transaction management were not addressed in this thesis. Some open problems in transaction management in object–based systems are discussed below. The definition of a transaction model and concurrency control algorithms introduces many areas where research is required. This discussion does not attempt to provide direction for the research but to identify interesting problems that remain.

A prototype of the algorithms should be developed so that performance studies can be made. The implementation of the algorithms will provide insight into the techniques employed here and provide direction in the design of future concurrency control algorithms. Various situations can be implemented to analyze the algorithms and identify the suitability of the algorithms for the variety of applications envisioned for object–based systems.

The algorithms introduced do not consider reliability and recovery issues. The use of nesting and independent Object Schedulers increases the complexity in ensuring reliable executions. The recovery techniques required to support reliable executions as failures occur are more complex due to the increased amount of information that must be stored (relative to object local and global information) and the complexity of the nested executions. In addition, this thesis does not consider deadlocks. Any blocking protocol is subject to deadlock and the increase in the amount of executing transactions in a nested environment leads to a greater chance of deadlock. A study of the types of deadlock that can occur and the means of handling them, similar to that of Härder and Rothermel [14] for nested transactions, is required for object–based systems. Both of these areas require a large amount of research and should provide for many interesting results.

Finally, the algorithms must be considered within a distributed environment. This thesis focuses on a centralized object–based system but the popularity of distribution requires the design of algorithms for a distributed object–based system. The algorithms developed in this thesis begin to support distribution through the use of schedulers at each object. This allows for an object to be placed at any site within a distributed system since its scheduler will be placed with it. This requires a mechanism for locating an object and relaying messages across a network so that the Method Scheduler can communicate with the Object Schedulers. Research must also address the distribution of the Method Scheduler. Many problems can occur in a distributed environment if the Method Scheduler were to reside on one node of a system [23]. Therefore, the Method Scheduler's algorithm must be redesigned so that its functionality can execute efficiently when distributed across a number of nodes. Demand for distributed systems is increasing and results from research into a distributed object–based system should prove to be very interesting.

Bibliography

[1] D. Agrawal and A. El Abbadi. A non-restrictive concurrency control for object oriented databases. In *Proceedings of the International Conference on Extending Database Technology*, pages 469–482, 1992.

[2] B.R. Badrinath and K. Ramamritham. Synchronizing transactions on objects. *IEEE Transactions on Computers*, 37(5):541–547, 1988.

[3] K. Barker, M. Evans, R. McFadyen, and K. Periyasamy. A formal ontological object–oriented model. Technical Report 92-02, University of Manitoba, 1992.

[4] C. Beeri, P.A. Bernstein, and N. Goodman. A model for concurrency in nested transactions systems. *Journal of the Association for Computing Machinery*, 36(2):230–269, 1989.

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

[6] E. Bertino and L. Martino. Object-oriented database management systems: Concepts and issues. *IEEE Computer*, 24(4):33–47, 1991.

[7] M. Cart, J. Ferrié, and J.F. Pons. Object modeling when using a multi-level transaction model. In *Proceedings of the Workshop on Transactions and Objects*, pages 41–47, 1990. Held in conjunction with OOPSLA/ECOOP 90.

[8] O. Deux *et al.* The story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, 1990.

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

[10] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 1987.

[11] J.F. Garza and W. Kim. Transaction management in an object-oriented database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 37–45. ACM, 1988.

[12] P. Graham, M. Zapp, and K. Barker. Applying method data dependence to transactions in object bases. Technical Report 92-07, University of Manitoba, 1992.

[13] T. Hadzilacos and V. Hadzilacos. Transaction synchronisation in object bases. *Journal of Computer and System Sciences*, 43(1):2–24, 1991.

[14] T. Härder and K. Rothermel. Concurrency control issues in nested transactions. *The VLDB Journal*, 2(1):39–74, 1993.

[15] M. Herlihy. Extending multiversion time-stamping protocols to exploit type information. *IEEE Transactions on Computers*, C-36(4):443–448, 1987.

[16] M. Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Transactions on Database Systems*, 15(1):96–124, 1990.

[17] M.P. Herlihy and W. Weihl. Hybrid concurrency control for abstract data types. *Journal of Computer and System Sciences*, 43(1):25–61, 1991.

[18] S.E. Hudson and R. King. Cactis: A self-adaptive, concurrent implementation of an object-oriented database management system. *ACM Transactions on Database Systems*, 14(3):291–321, 1989.

[19] W. Kim. Object-oriented databases: Definition and research directions. *IEEE Transactions on Knowledge and Data Engineering*, 2(3):327–341, 1990.

[20] W. Kim, J.F. Garza, N. Ballou, and D. Woelk. Architecture of the ORION next-generation database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109–124, 1990.

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

[22] P. Muth, T.C. Rakow, W. Klas, and E.J. Neuhold. A transaction model for an open publication environment. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 160–218. Morgan Kaufmann, 1992.

[23] M.T. Özsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, 1991.

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

[25] T.C. Rakow, J. Gu, and E.J. Neuhold. Serializability in object-oriented database systems. In *Proceedings of the International Conference on Data Engineering*, pages 112–120. IEEE, 1990.

[26] R.F. Resende and A. El Abbadi. A graph testing concurrency control protocol for object bases. In *Proceedings of the International Conference on Computing and Information*, pages 289–292, 1992.

[27] P.M. Schwarz and A.Z. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3):223–251, 1984.

[28] W. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, 1988.

[29] W. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–282, 1989.

[30] W. Weihl and B. Liskov. Implementation of resilient, atomic data types. *ACM Transactions on Programming Languages and Systems*, 7(2):244–269, 1985.

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

[32] G. Weikum and H.J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 516–553. Morgan Kaufmann, 1992.

[33] K. Wilkinson, P. Lyngbæk, and W. Hasan. The Iris architecture and implementation. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):63–75, 1990.

[34] M. Zapp and K. Barker. An architecture and model for transactions in object bases. Technical Report 92-09, University of Manitoba, 1992.

[35] M. Zapp and K. Barker. The serializability of transactions in object bases. In *Proceedings of the International Conference on Computing and Information*, 1993.