Trigger Management in Active Multidatabase Systems

by

William Douglas Baker

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

Winnipeg, Manitoba, Canada, 1994

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Manitoba to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

ISBN 0-315-99087-2

Name Bill Baker

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
SUBJECT CODE

U·M·I

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

Nom _____

Dissertation Abstracts International est organisé en catégories de sujets. Veuillez s.v.p. choisir le sujet qui décrit le mieux votre thèse et inscrivez le code numérique approprié dans l'espace réservé ci-dessous.

		U·M·I
SUJET		CODE DE SUJET

Catégories par sujets

HUMANITÉS ET SCIENCES SOCIALES

COMMUNICATIONS ET LES ARTS

Architecture 0729
Beaux-arts 0357
Bibliothéconomie 0399
Cinéma 0900
Communication verbale 0459
Communications 0708
Danse 0378
Histoire de l'art 0377
Journalisme 0391
Musique 0413
Sciences de l'information 0723
Théâtre 0465

ÉDUCATION

Généralités 515
Administration 0514
Art 0273
Collèges communautaires 0275
Commerce 0688
Économie domestique 0278
Éducation permanente 0516
Éducation préscolaire 0518
Éducation sanitaire 0680
Enseignement agricole 0517
Enseignement bilingue et multiculturel 0282
Enseignement industriel 0521
Enseignement primaire. 0524
Enseignement professionnel 0747
Enseignement religieux 0527
Enseignement secondaire 0533
Enseignement spécial 0529
Enseignement supérieur 0745
Évaluation 0288
Finances 0277
Formation des enseignants 0530
Histoire de l'éducation 0520
Langues et littérature 0279
Lecture 0535
Mathématiques 0280
Musique 0522
Orientation et consultation 0519
Philosophie de l'éducation 0998
Physique 0523
Programmes d'études et enseignement 0727
Psychologie 0525
Sciences 0714
Sciences sociales 0534
Sociologie de l'éducation 0340
Technologie 0710

LANGUE, LITTÉRATURE ET LINGUISTIQUE

Langues
- Généralités 0679
- Anciennes 0289
- Linguistique 0290
- Modernes 0291

Littérature
- Généralités 0401
- Anciennes 0294
- Comparée 0295
- Médiévale 0297
- Moderne 0298
- Africaine 0316
- Américaine 0591
- Anglaise 0593
- Asiatique 0305
- Canadienne (Anglaise) 0352
- Canadienne (Française) 0355
- Germanique 0311
- Latino-américaine 0312
- Moyen-orientale 0315
- Romane 0313
- Slave et est-européenne 0314

PHILOSOPHIE, RELIGION ET THÉOLOGIE

Philosophie 0422
Religion
- Généralités 0318
- Clergé 0319
- Études bibliques 0321
- Histoire des religions 0320
- Philosophie de la religion 0322

Théologie 0469

SCIENCES SOCIALES

Anthropologie
- Archéologie 0324
- Culturelle 0326
- Physique 0327

Droit 0398
Économie
- Généralités 0501
- Commerce-Affaires 0505
- Économie agricole 0503
- Économie du travail 0510
- Finances 0508
- Histoire 0509
- Théorie 0511

Études américaines 0323
Études canadiennes 0385
Études féministes 0453
Folklore 0358
Géographie 0366
Gérontologie 0351
Gestion des affaires
- Généralités 0310
- Administration 0454
- Banques 0770
- Comptabilité 0272
- Marketing 0338

Histoire
- Histoire générale 0578
- Ancienne 0579
- Médiévale 0581
- Moderne 0582
- Histoire des noirs 0328
- Africaine 0331
- Canadienne 0334
- États-Unis 0337
- Européenne 0335
- Moyen-orientale 0333
- Latino-américaine 0336
- Asie, Australie et Océanie 0332

Histoire des sciences 0585
Loisirs 0814
Planification urbaine et régionale 0999
Science politique
- Généralités 0615
- Administration publique 0617
- Droit et relations internationales 0616

Sociologie
- Généralités 0626
- Aide et bien-être social 0630
- Criminologie et établissements pénitentiaires 0627
- Démographie 0938
- Études de l' individu et de la famille 0628
- Études des relations interethniques et des relations raciales 0631
- Structure et développement social 0700
- Théorie et méthodes. 0344
- Travail et relations industrielles 0629

Transports 0709
Travail social 0452

SCIENCES ET INGÉNIERIE

SCIENCES BIOLOGIQUES

Agriculture
- Généralités 0473
- Agronomie. 0285
- Alimentation et technologie alimentaire 0359
- Culture 0479
- Élevage et alimentation 0475
- Exploitation des pâturages 0777
- Pathologie animale 0476
- Pathologie végétale 0480
- Physiologie végétale 0817
- Sylviculture et faune 0478
- Technologie du bois 0746

Biologie
- Généralités 0306
- Anatomie 0287
- Biologie (Statistiques) 0308
- Biologie moléculaire 0307
- Botanique 0309
- Cellule 0379
- Écologie 0329
- Entomologie 0353
- Génétique 0369
- Limnologie 0793
- Microbiologie 0410
- Neurologie 0317
- Océanographie 0416
- Physiologie 0433
- Radiation 0821
- Science vétérinaire 0778
- Zoologie 0472

Biophysique
- Généralités 0786
- Medicale 0760

SCIENCES DE LA TERRE

Biogéochimie 0425
Géochimie 0996
Géodésie 0370
Géographie physique 0368
Géologie 0372
Géophysique 0373
Hydrologie 0388
Minéralogie 0411
Océanographie physique 0415
Paléobotanique 0345
Paléoécologie 0426
Paléontologie 0418
Paléozoologie 0985
Palynologie 0427

SCIENCES DE LA SANTÉ ET DE L'ENVIRONNEMENT

Économie domestique 0386
Sciences de l'environnement 0768
Sciences de la santé
- Généralités 0566
- Administration des hôpitaux 0769
- Alimentation et nutrition 0570
- Audiologie 0300
- Chimiothérapie 0992
- Dentisterie 0567
- Développement humain 0758
- Enseignement 0350
- Immunologie 0982
- Loisirs 0575
- Médecine du travail et thérapie 0354
- Médecine et chirurgie 0564
- Obstétrique et gynécologie 0380
- Ophtalmologie 0381
- Orthophonie 0460
- Pathologie 0571
- Pharmacie 0572
- Pharmacologie 0419
- Physiothérapie 0382
- Radiologie 0574
- Santé mentale 0347
- Santé publique 0573
- Soins infirmiers 0569
- Toxicologie 0383

SCIENCES PHYSIQUES

Sciences Pures

Chimie
- Généralités 0485
- Biochimie 487
- Chimie agricole 0749
- Chimie analytique 0486
- Chimie minérale 0488
- Chimie nucléaire 0738
- Chimie organique 0490
- Chimie pharmaceutique 0491
- Physique 0494
- PolymÇres 0495
- Radiation 0754

Mathématiques 0405
Physique
- Généralités 0605
- Acoustique 0986
- Astronomie et astrophysique 0606
- Electronique et électricité 0607
- Fluides et plasma 0759
- Météorologie 0608
- Optique 0752
- Particules (Physique nucléaire) 0798
- Physique atomique 0748
- Physique de l'état solide 0611
- Physique moléculaire 0609
- Physique nucléaire 0610
- Radiation 0756

Statistiques 0463

Sciences Appliqués Et Technologie

Informatique 0984
Ingénierie
- Généralités 0537
- Agricole 0539
- Automobile 0540
- Biomédicale 0541
- Chaleur et ther modynamique 0348
- Conditionnement (Emballage) 0549
- Génie aérospatial 0538
- Génie chimique 0542
- Génie civil 0543
- Génie électronique et électrique 0544
- Génie industriel 0546
- Génie mécanique 0548
- Génie nucléaire 0552
- Ingénierie des systèmes 0790
- Mécanique navale 0547
- Métallurgie 0743
- Science des matériaux 0794
- Technique du pétrole 0765
- Technique minière 0551
- Techniques sanitaires et municipales 0554
- Technologie hydraulique 0545

Mécanique appliquée 0346
Géotechnologie 0428
Matières plastiques (Technologie) 0795
Recherche opérationnelle 0796
Textiles et tissus (Technologie) 0794

PSYCHOLOGIE

Généralités 0621
Personnalité 0625
Psychobiologie 0349
Psychologie clinique 0622
Psychologie du comportement 0384
Psychologie du développement 0620
Psychologie expérimentale 0623
Psychologie industrielle 0624
Psychologie physiologique 0989
Psychologie sociale 0451
Psychométrie 0632

TRIGGER MANAGEMENT IN ACTIVE MULTIDATABASE SYSTEMS

BY

WILLIAM DOUGLAS BAKER

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

The University of Manitoba requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Multidatabase systems are constructed from autonomous independent database managers and are an alternative to homogeneous integrated distributed database systems. Two or more data items that are related through a value dependency are termed interdependent data. Multidatabase systems may require that the value dependencies of interdependent data be satisfied. Active databases define rules or triggers to automatically perform actions when certain database conditions arise. This thesis provides a model for an active multidatabase system capable of maintaining interdependent data. It rigorously formulates the different types of triggers and events that participate in an active multidatabase system. A definition of active multidatabase serializability (called AMDB-serializability) is introduced and a graph theoretic tool is described that defines precisely when a given execution sequence is serializable. An architecture for trigger and transaction management in an active multidatabase system is defined to describe the interactions between the different software layers of the management facilities. The architecture is used to identify the components that are required for trigger management, event detection, and concurrency control. This thesis introduces algorithms for these components and describes the interactions between them. The correctness of the concurrency control algorithms in ensuring AMDB-serializability is shown using the graph theoretic tool. These algorithms are incorporated in such a way that they do not violate the autonomy of the local database systems.

Acknowledgements

I would like to thank many people for their support while I was writing this thesis. First, and foremost, I would like to thank my supervisor Dr. Ken Barker. He taught me how to do research and how to write up your results. There often were times when I questioned the directions he was taking me but I now understand that all of the "grunt work" was necessary to ensure the research is well-founded. Second, I would like to thank the members of the Advanced Database Systems Laboratory. The research group meetings that were held formed new ideas that were eventually added to this work. Third, I must thank the good people of the Computer Science and Math department at Brandon University. They gave me a strong background in the undergraduate principles upon which this thesis rests. Finally, I would like to thank my parents. Their encouragement and support over these many years cannot be matched and I could not have done it without them.

Contents

List of Figures

Glossary of Acronyms

AGS	Active Global Scheduler
AGSS	Agressive Global Serial Scheduler
AMDB	Active Multidatabase
AMDBSR	Active Multidatabase Serializable
AMDMS	Active Multidatabase Management System
AMDS	Active Multidatabase System
AMSG	Active Multidatabase Serializability Graph
CDG	Causal Dependency Graph
DG	Dependency Graph
EL	Event Log
GH	Global History
GRM	Global Recovery Manager
GS	Global Scheduler
GSH	Global Subtransaction History
GSS	Global Serial Scheduler
GST	Global Subtransaction
GT	Global Transaction
GTF	Global Transaction Family
GTM	Global Transaction Manager
GTR	Global Trigger
GTRD	Global Trigger Dictionary
GTRM	Global Trigger Manager
LDB	Local Database
LH	Local History
LRM	Local Recovery Manager
LS	Local Scheduler
LT	Local Transaction
LTM	Local Transaction Manager
MDBSR	Multidatabase Serializability
MDMS	Multidatabase Management System
MSG	Multidatabase Serializability Graph
TFT	Transaction Family Tree

Chapter 1

Introduction

A *multidatabase system (MDB)* is a collection of autonomous and possibly heterogeneous, pre-existing local database systems (DBMSs). An MDB supports global applications that access data items in more than one local database. This environment differs from traditional homogeneous distributed database systems in that it interconnects DBMSs in a bottom up fashion, thereby allowing existing applications developed on each of the DBMSs to be executable without modification. An MDB allows the sharing of data and resources at the same time as retaining the autonomy of local database systems.

The autonomy of the local systems makes multidatabase research especially difficult. This requirement means that any work done to the MDS cannot modify the local database systems local operations. That is, a local database participating in an MDS has its choice of transaction model, query techniques, hardware configurations, etc.

A high level component view of an MDS is presented in Figure 1.1. The environment contains a number of independent local database systems and the MDMS

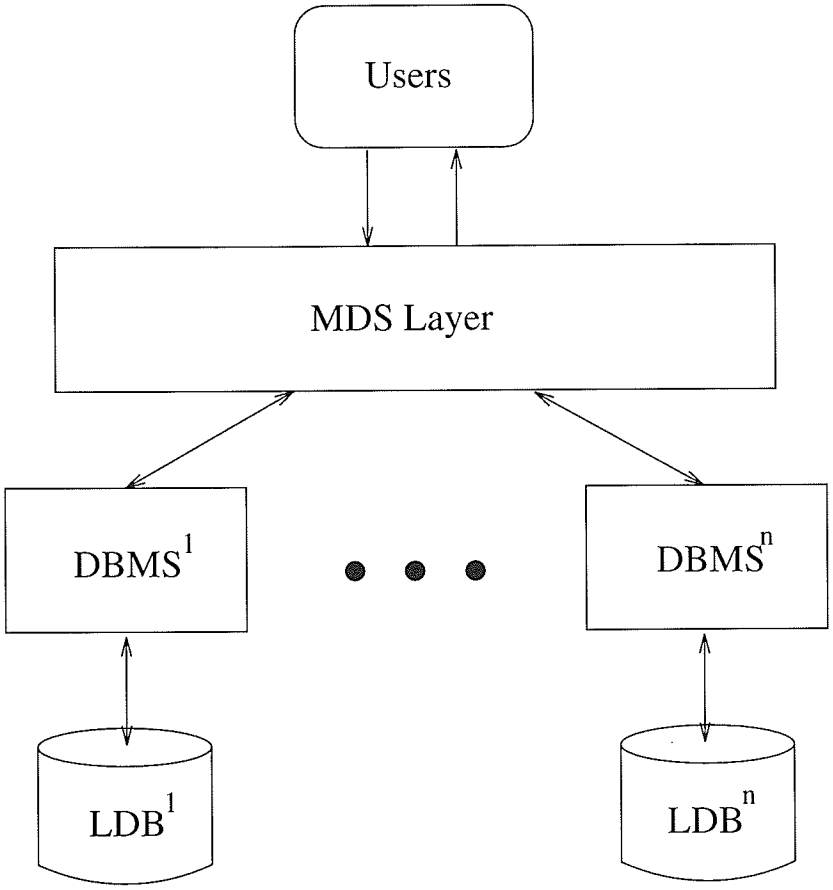

Figure 1.1: Components of an MDS

layer which lies above them. The MDMS allows users to access multiple local databases with one transaction. Local users of some database may also submit transactions directly, as if the local database did not exist as part of an MDS. As far as a local database is concerned the MDMS is just another local user. This construction maintains the autonomy of each local database.

The areas of research in multidatabase systems include schema integration, aspects of heterogeneity, autonomy, differences in data representation, global query processing, concurrency control, reliability, security, and global integrity constraints.

The aspects of global integrity constraints is particularly relevant to this thesis so we describe it in more detail in the next section.

1.1 Interdatabase Dependencies

One of the problems in multidatabase systems is the maintenance of global integrity constraints. These are often called *interdatabase dependencies.* Interdatabase dependencies arise when two or more data items located at different local databases are related in some way such that if one is updated with a new value then the other may have to be updated in order for the two data items to be semantically correct. The following presents some examples of interdatabase dependencies.

Example 1.1.1 Consider a multidatabase system consisting of a phone company database and a construction company database. The two associations are working together on several projects and require access to each others data. The information for one of the projects is read often by users at each site but is rarely updated. The database administrators have decided to replicate this information at each site to improve performance. Hence, we have the following interdatabase dependency:

$$Project\_X\_copy1 == Project\_X\_copy2$$

If some transaction updates one of the project information items then the system must automatically update the other.

The company leaders have also decided that the cost of project X must never exceed the cost of project Y. The phone company manages the cost of project X and the construction company manages the cost of project Y. This results in a second interdatabase dependency:

$$Project\_X\_cost \leq Project\_Y\_cost$$

■

1.2 Active Database Systems

Traditional databases are passive persistent stores. Data in the database are inserted, deleted, modified, and otherwise manipulated. The database does not perform operations so all operations on the data are made entirely by users including constraining their values. It is the users responsibility to ensure operations on data are "semantically correct" and that all necessary actions are performed, including maintaining integrity constraints.

Active databases define *rules* or *triggers* to automatically perform actions when certain conditions in the database arise. These actions usually occur as a response to actions caused by transactions executing on the database. The database monitors transaction execution, detects events that trigger actions, sets up triggered actions, and executes them.

Active databases and knowledge bases use similar constructs [35]. Both react to events generated in the database and respond with some action. However, the main difference between the two is the way triggers are executed. Database triggers are executed as a result of normal database operations. Rules in knowledge bases are usually executed upon explicit request of an application and attempt to derive information from a database of *facts*.

Triggers have many uses in active databases including maintaining integrity constraints, view management, access control, logging, alerting, etc. This helps in writing database tasks by removing the responsibility of executing consistency

maintaining operations for the user. In addition, triggers are only defined once so code reuse occurs and fewer errors are experienced.

Research in active databases include detection of complex events, trigger execution models, trigger correctness, optimization of trigger execution, etc. The features of active databases are currently being extended to other paradigms such as object-oriented databases [1, 19, 16, 7].

1.2.1 Active Multidatabase Systems

We define an *active multidatabase system (AMDS)* that will automatically perform actions when certain "multidatabase conditions" arise. Traditional active database use *event detection* mechanisms to monitor the operations of executing transactions. Each event is generally passed to a trigger subsystem which determines if a trigger event has occurred, and if so, will execute the triggered action. The nature of a multidatabase system complicates trigger event detection and execution because a multidatabase is constructed in a "bottom-up" fashion where local database systems are autonomous and cannot be modified. If a local database system is not initially active then detecting events of executing multidatabase transactions at that site is very difficult. Local databases that are initially active would require extensive modifications to allow the multidatabase to benefit from the trigger execution and event detection techniques. Mechanisms that incorporate activeness into a multidatabase system must ensure that the autonomy of the local systems is not violated.

1.3 AMDS Architecture

We use an active multidatabase architecture (AMDS) as a platform for the work presented in this thesis. The components of an AMDS are depicted in Figure 1.1. Each local database is managed by a different DBMS. The AMDMS layer provides users with the ability to access different local databases. We will concentrate on the particular events necessary for trigger and transaction management in this environment (Figure 1.2). This architecture assumes that each local database system has a common centralized structure (for example, Bernstein *et al.* [8]) and is based on one provided by Barker [6].

The components of a DBMS are a *Local Transaction Manager* (LTM), a *Local Scheduler* (LS), and a *Local Recovery Manager* (LRM). The LTM interacts with the user, coordinates the atomic execution of transactions, and handles transaction initialization procedures such as transaction identification. The LS is responsible for correct concurrent execution of transactions submitted to the DBMS and may use any concurrency control technique. The Local Recovery Manager ensures the local database (LDB) contains precisely the effects of committed transactions.

The AMDMS layer rests above the DBMSs and provides a communication facility between local DBMSs. Its four components include a *Global Transaction Manager* (GTM), *Global Trigger Manager* (GTRM), *Global Scheduler* (GS), and a *Global Recovery Manager* (GRM). The GTM handles transaction management duties related to transactions spanning multiple databases including accepting transaction submission from users, ensuring their syntactic correctness, and ultimately returning results to the user. The Global Trigger Manager is responsible for controlling event detection and trigger execution in the multidatabase environment. The Global Scheduler provides concurrency control for global transactions and multi-

database triggers. The Global Recovery Manager ensures that global transactions and triggers execute reliably and are recoverable.

1.4 Problem Definition

This thesis addresses the problem of trigger management and concurrent transaction execution in active multidatabase systems. *Active multidatabase serializability* is defined, which is an extension of *multidatabase serializability* [6] applicable to multidatabase systems. This provides a theoretical basis for the discussion of concurrency control algorithms. Event detection and trigger execution mechanisms are described. The thesis then presents a scheduler for executing transactions and triggered actions concurrently in an active multidatabase environment.

This thesis makes the following contributions:

1. Provides a summary of previous research related to *active multidatabase systems.*

2. Discusses the problems encountered in providing general trigger management in an active multidatabase system.

3. Introduces a new correctness criteria for the active multidatabase environment.

4. Provides mechanisms for event detection and trigger support with minimal violation of local database autonomy.

5. Offers a global scheduling algorithm which guarantees correct execution of global transactions and triggered transactions.

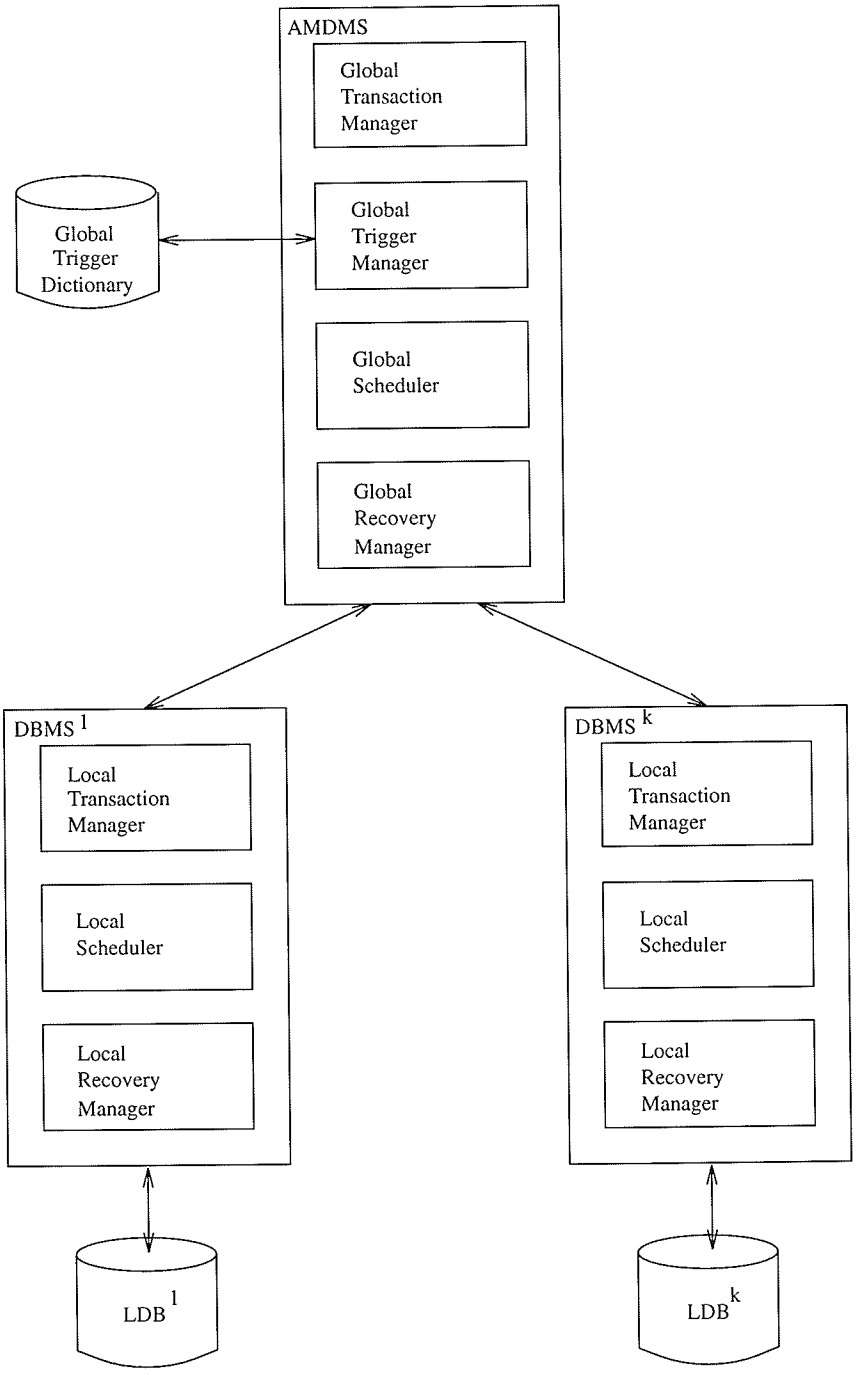

Figure 1.2: AMDS Architecture

6. Proves that this algorithm is correct.

This is then used to illustrate how global integrity constraints can be supported as described in Section 1.1. Therefore, the constraint problem can be seen as both the motivation for this work and the final challenge that will demonstrate the utility of an active multidatabase system.

1.5 Outline of Thesis

The remainder of this thesis is as follows. Chapter 2 discusses fundamental transaction definitions and presents previous related work. A formal transaction and trigger model and the study of AMDB-serializability are provided in Chapter 3. Chapter 4 discusses event detection and trigger execution in this environment. A concurrency control algorithm is presented in Chapter 5 and its correctness is proved. Finally, Chapter 6 summarizes the thesis and presents some suggestions for future research.

Chapter 2

Background

This chapter presents the necessary background information. Relevant research includes active databases, transaction management (specific to multidatabase systems), and interdependent data management of multidatabases.

2.1 Fundamental Definitions

The definitions and notations for traditional transaction management are taken from Özsu and Valduriez [28]. $O_{ij}\epsilon$ {read,write} denotes operation j of transaction i. OS_i is the set of all operations of transaction i ($\bigcup_k O_{ik}$). We also denote with N_i the termination condition for T_i, where $N_i\epsilon$ {abort,commit}. The abbreviations r,w,a, and c will be used for the read, write, abort, and commit operations, respectively. A traditional transaction is:

Definition 2.1.1 *(Transaction):* A transaction T_i is a partial order $T_i = \{\Sigma_i, \prec_i\}$ where Σ_i is the domain consisting of the operations and the termination condition

of T_i, and $\prec_i$ is an irreflexive and transitive binary relation indicating the execution order of these operations such that

1. $\Sigma_i = OS_i \bigcup \{N_i\}$,
2. for any two operations O_{ij}, $O_{ik} \epsilon OS_i$, if $O_{ij} = r(x)$ and $O_{ik} = w(x)$ for any data item x, then either $O_{ij} \prec_i O_{ik}$ or $O_{ik} \prec_i O_{ij}$, and
3. $\forall O_{ij} \epsilon OS_i$, $O_{ij} \prec_i N_i$. ■

A *history* is a log that records the execution of transactions. Histories provide a means for inspecting the execution of transactions to ensure that they have a correct ordering when executed concurrently. Thus, histories are defined with respect to transactions:

Definition 2.1.2 *(History):* Given a DBMS with 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 $T_j \epsilon \mathcal{T}$,
2. $\prec_H \supseteq \bigcup_j \prec_j$ where $\prec_j$ is the ordering relation for transaction T_j at the DBMS, and
3. for any two conflicting operations p,q ϵH, either $p \prec_H q$ or $q \prec Hp$. ■

This definition implies that if the operations of two transactions conflict, the transactions also conflict.

2.1.1 Serializability

The generally accepted correctness criterion for traditional transactions is *conflict serializability*, where conflicting operations of transactions must be ordered so the transactions appear to execute serially. That is, all the operations of one transaction must appear to execute before all of the operations of another transaction. This correctness criterion requires the following definitions:

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

Definition 2.1.4 *(Conflict Equivalent:)* Two histories are *conflict equivalent* if they are over the same set of transactions and order conflicting operations identically. ■

These definitions lead to the definition of a serializable history:

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

2.2 Related Work

Polytransactions

Sheth *et al.* [34, 30, 32, 25, 33] define the *polytransaction model* for the management of interdependent data (data stored in two or more databases that are related through an integrity constraint). They propose a framework with two important

features: a declarative specification of interdependent data and their mutual consistency requirements, and use of the specification to automatically generate update transactions that manage interdependent data. The declarative specification involves collecting all of the dependencies and storing them together. This allows them to be examined and modified in one place, independent of application programs. Automating updates eliminates the error prone approach used when the transaction designer manages interdependent data. Various aspects of Sheth's model are discussed below.

Specification of Interdatabase Dependencies: The first feature must capture interdatabase dependencies. Sheth's model uses *Data Dependency Descriptors* (D^3) which is a 5-tuple:

$$D^3 = \text{<S,U,P,C,A>}$$

where:

S is the set of *source data objects*,

U is the *target data object*,

P is a boolean-valued predicate called the *interdatabase dependency predicate* (dependency component). It specifies a relationship between the source and target data objects, and evaluates to true if this relationship is satisfied.

C is a boolean-valued predicate, called the *mutual consistency predicate* (consistency component). It specifies consistency requirements and defines when P must be satisfied.

A is a collection of *consistency restoration procedures* (action component). Each procedure specifies actions that must be taken to restore consistency defined by P.

Data Dependency Descriptors are uni-directional; from the set of source data objects to the target data object. Performing an operation on the source or target data objects may require additional actions to maintain mutual consistency of interdependent data. The *Interdatabase Dependency Schema* (IDS) is the set of all D^3s enforced in a multidatabase system.

System Architecture The polytransaction model uses an architecture similar to that described in Section 1.3 [34] but requires some additional features. Each LDBS is augmented with a *dependency subsystem* (DS) which acts as an interface between LDBSs and can communicate with DSs of other LDBSs. A DS analyzes transactions submitted to its LDBS for updates related to data in other databases by consulting the IDS for D^3s involving data modified by the local transaction. If it discovers that the transaction updates related data in other databases, then a series of update transactions are scheduled.

Properties of Polytransactions A Polytransaction (T^+) is the "transitive closure" of a transaction T submitted to the multidatabase management system. The transitive closure is computed with respect to the IDS [34]. That is, a polytransaction is a nested transaction with the original transaction T as root and any triggered transactions as children. For each data item X modified by T the IDS must be checked. If there exists a D^3 with X as a source data object and the effects of T violate that D^3s dependency and consistency predicates then a new subtransaction T' must be created to update the target data object. A new node

corresponding to T' is added to the tree as a child of the T node. This process is then applied iteratively to all children of T. These subtransactions correspond to the consistency restoration procedures defined in the D^3s.

Children are related to their parents in two ways which are specified by the coupling mode in the consistency restoration procedures of the D^3s.

1. The subtransaction may be *coupled* (where the parent waits for the child to complete).

2. The subtransaction may be *decoupled* (so it runs concurrently). If it is coupled then it may be either *vital* (the parent fails if the child fails) or *non-vital.*

Interdatabase Dependency Schema The dependency predicate P is a boolean-valued expression specifying the relationship that should hold between the source and target data objects [30]. It can be specified using the traditional operators of relational algebra (selection, projection, join, difference, union, intersection, *etc.*) as well as aggregate (ξ) and transitive closure (α) operators.

The mutual consistency requirement specifies "how far" the interdependent data is allowed to diverge before the consistency must be restored. The specification of the consistency predicate can involve multiple boolean valued terms called *consistency terms* and each is denoted by c_i. C is a logical expression constructed using the c_is and the logical operators $\wedge$, $\vee$ and $\neg$. The predicate can be defined both in terms of time and data state.

A few examples will prove useful (see Sheth *et al.* [34] for details):

- @11:00 on 21-Nov-1991 means the data must be consistent at 11 o'clock on the 21^{st} of November, 1991.

- 25-Aug-1992 ! means data must be consistent after August 25, 1992.
- $\Delta EMP.Salary > 500$ specifies that the data must be consistent if a transaction changes an employee's salary by more than $500.
- 10 updates on R_1 where $R_1 \epsilon$ S specifies that data must be consistent after the 10^{th} update to R_1, a source object.

When a D^3's dependency and consistency predicates have been violated, the set of consistency restoration procedures must be used to generate appropriate update transactions. The authors assume the existence of a *multidatabase monitor* which examines updates to databases for violations of D^3s in the IDS. If a violation occurs then the monitor must use the consistency restoration procedure of the violated D^3 to generate an update transaction. There may be several ways to violate the predicates of a D^3 so there may be several consistency restoration procedures.

OSCA

Srinidhi [31] describes an interoperability architecture called OSCA that provides large corporations the flexibility to combine software products in ways which satisfy authorized user needs. OSCA supports data redundancy as a means for achieving improved performance, reliability, and availability.

OSCA separates business functionality into three "layers" of :

- corporate data management functionality (*data layer*),
- business aware operations and management functions (*processing layer*), and
- human interaction functionality (*user layer*).

Each layer is partitioned into autonomous units called *building blocks* that contain software to implement each layer's functionality. The data layer contains *data layer building blocks* (DLBBs), the processing layer contains *processing layer building blocks* (PLBBs), and the user layer contains *user layer building blocks* (ULBBs). Each DLBB *stewards* some allocated portion of the corporate data so it is completely responsible for the data that it stewards.

Building blocks interact with each other using interfaces called *contracts*. *Contract specifications* define the way functionality provided by a contract is invoked and any support commitments that are required. Contract specifications include functionality, interface, response time, and availability.

Redundant data in OSCA falls into one of two categories. *Private redundant data* are copies of stewarded data owned within individual building blocks but are not visible outside of the building block. *Shared redundant data* are supported only in a DLBB and outside the building block they are visible for retrieval only.

If a building block has private redundant data and requires updates to be sent by the stewarding DLBB whenever the stewarding data is updated then pre-defined create/update/delete (CUD) contracts have to be established between the two. If a DLBB supports shared redundant data then it offers retrieval contracts on that data. Pre-defined CUD contracts must also be established with the stewarding DLBB.

OSCA provides two schemes for updating redundant data. The Linked Contracts Table Scheme is used when eventual or lagging consistency of redundant data is acceptable. Each redundant copy requiring updates is responsible for establishing the update frequency with the stewarding DLBB. The update frequency, contracts for updating redundant data, and alternate data views are all stored in the *linked contracts table* which may be part of the stewarding DLBB or managed

by *redundancy management services* (RMS).

When an update occurs at the stewarding DLBB it commits the update and provides RMS with the results. The RMS consults the linked contracts table to determine which redundant copies should get the update and provides *contract interaction services* (CIS) with the information. The CIS then invokes the corresponding update contract at DLBBs containing the redundant data. Since the stewarding DLBB commits the results of the update before any update checks are made this technique will only ensure eventual or lagging consistency.

The Synchronous Update Scheme is useful if the redundant and original data must be the same at all times. This technique requires that updates to stewarded data be committed after the corresponding redundant data is updated so the stewarding DLBB provides the RMS with the update prior to committing the result. The RMS consults the table for redundant copies that require synchronous update and provides CIS with the results. CIS uses transaction monitors (TMs) to invoke a distributed transaction across building blocks which require synchronous updates to the redundant data. The TM facilities must use two-phase commit to ensure correct execution.

Quasi-transactions + Rules

Arizio *et al* [5] describe a rule model and system architecture to manage inter-database dependencies in a multidatabase system using quasi-transactions. Their rule model follows the HiPAC model for active database systems. A rule has the following structure: (OL,OM,E,C,A). OL are the data read by the rule and OM are the data written, inserted, or deleted by the operation described in A. E is the event predicate that describes when the dependency has to be verified. It can be a simple event or a complex combination of events using *and*, *or*, or *not*. C describes

the condition predicate that must hold among the interdependent data of the rule. A is the action component of the rule which restores consistency when executed.

A quasi-transaction (QT) consists of a set of operations, each of which may be basic data operations or another QT. A quasi-transaction defines weak ACID properties. The inclusion of a QT and its descendent QTs is described as a tree. A parent QT terminates when all of its children terminate. In case of failure, QTs offer a mechanism to repair actions thereby modifying the outcome to a successful one.

Quasi-transactions may be scheduled in three kinds of ways: immediate, deferred, and coupled. Immediate scheduling suspends execution of the parent until the child completes. Deferred scheduling delays the QT so it is executed as the last child of the parent. Decoupled scheduling starts a QT that has no termination relationship with the creator. They are executed as siblings of the creator. The HUC (hold until commit) option is used with decoupled QTs if it is to be executed only if the creator commits.

The model also supports transactions that require full ACID properties. They may be nested and their children must be transactions. A child may commit but its results will be committed only if its parent commits. If a transaction aborts, its children are aborted too, but a subtransaction may abort without causing the abortion of its parent.

Transactions can also be executed as immediate, deferred, or decoupled. However, the behavior of the deferred and decoupled modes differs from proper quasi-transactions. Deferred transactions are deferred until the commit of the entire transaction subtree in which they are generated. Hence, they become children of the root transaction and are executed concurrently just before the root's commit. A decoupled transaction begins a completely new transaction, independent from

the one that created it. The decoupled transaction becomes a sibling of the root transaction of the subtree that created it.

Quasi-transaction exceptions provide a way of returning error information about success or failure from children QTs. Error conditions propagate up the QT tree until they reach a node with an appropriate handler for that exceptional condition. Error conditions that are not handled may cause the entire QT tree to abort.

A quasi-transaction has two types of error handlers. *External handlers* are those used to handle an exception arising from the QT itself. *Internal handlers* treat error conditions that have propagated up from a nodes children. Internal handlers may deal with child aborts and make decisions as to the aborts criticality.

This model assumes a system architecture where each local database consists of some commercial DBMS with a rule manager and rule base built onto it. The rule base contains the rules which are firable by transactions at the DBMS. The OL and OM clauses of rules are used to determine this. The Rule Manager consists of the Event Detector and the Controller. The Event Detector is able to recognize events caused by transaction operations as well as temporal events using infrastructural timing services. The Event Detector searches for rules that may fire because of event occurrences and passes these to the Controller. The Controller evaluates the condition portion of these rules and if the expression "E and (not C)" evaluates to true then the trigger action is executed according to its execution mode.

HiPAC

The HiPAC project is investigating active, time-constrained database management. HiPAC proposes an Event-Condition-Action (ECA) formalism for active databases capable of automatic enforcement of integrity constraints, expressing rela-

tionships between data items, alerters, access constraints, and triggers. These functions were previously implemented with special purpose mechanisms [14, 15, 24, 26].

HiPAC's object-oriented data model's rules are first-class objects. Rule attributes include:

Event The event that triggers the rule.

Condition A set of queries that are evaluated when the rule is triggered. Queries are specified in an object-oriented data manipulation language.

Action An operation that is executed when the rule is triggered if the condition is satisfied. The action could be database operations or calls to application programs.

E-C coupling A coupling mode that specifies when the condition is evaluated relative to the transaction in which the triggering event is detected.

C-A coupling Specifies when the action is executed relative to the transaction in which the condition is evaluated.

The event can be a primitive event such as a database operation (data definition, data manipulation), a temporal event (absolute, relative to another event, periodic), or an external notification (application defined event). Since database operations are not instantaneous it is possible to define two events for each operation: the *beginning* and the *end* of the operation.

The model also supports *composite* events. The *disjunction* of two events, E_1 and E_2, is a composite event signaled when either E_1 or E_2 is signaled. The *sequence* of two events, E_1 and E_2, is a composite event signaled when E_2 is signaled provided

that E_1 has already been signaled. The *closure* of event E occurs after E has been signaled an arbitrary number of times in a transaction.

E-C coupling modes can be one of the following:

1. *Immediate:* Evaluate the condition immediately within the context of the transaction causing the event.
2. *Deferred:* Evaluate the condition upon termination of the transaction causing the event, but before it commits.
3. *Separate, causally dependent:* Evaluate the condition in a separate transaction after and iff the triggering transaction commits.
4. *Separate, causally independent:* Evaluate the condition in a separate transaction. The scheduler may schedule this transaction independent of the triggering transaction.

C-A coupling have precisely the same coupling modes except the relationship exists between the condition and the action.

Rules have special operations of their own:

Fire Evaluate the condition and if satisfied execute the action.

Disable Disable the automatic rule firing for the event.

Enable Enable automatic rule firing for the event.

HiPAC uses a nested transaction model. When a rule fires, a new transaction is created and the rule's condition is evaluated by the spawned transaction. If the rule's E-C coupling is immediate then the triggering transaction is suspended and

the spawned transaction is executed as a subtransaction. A deferred E-C coupling causes the condition transaction to be executed just prior to the parent's commit. If the E-C coupling is separate then the new transaction is a top-level transaction. It may be scheduled concurrently if it is causally independent.

If the rule's condition is satisfied then another transaction is created for the action and is dealt with in an analogous manner based on the C-A coupling of the triggered rule.

If the triggering transaction causes multiple rules to fire then a new transaction is created for each rule triggered. Immediate subtransactions will execute concurrently. Separate, causally independent transactions can be executed concurrently. Separate, causally dependent transactions will be scheduled after the triggering transaction has committed. If the execution of a rule's action causes more rules to be triggered then this process repeats itself, thereby creating a tree of nested transactions.

For example, a HiPAC rule such as:

Event:	Update Xerox price
Condition:	Where new price = 50
Action:	Send request to buy 500 shares for client A
E-C Coupling:	Separate, Causally dependent
C-A Coupling:	Immediate

This rule ensures that if the Xerox price is updated to 50 then 500 shares are bought for client A [26].

For a DBMS to support HiPAC's knowledge and execution models it must support nested transactions and object-oriented data management. Its functional components include:

Object Manager: Provides object-oriented data management.

Transaction Manager: Provides nested transactions.

Event Detectors: Detect events and signal them to the Rule Manager.

Rule Manager: Maps events to rule firings, and rule firings to transactions.

Condition Evaluator: Evaluates rule conditions.

Ode

Ode is an object-oriented database system and environment developed at AT&T Bell Labs [1, 19, 20]. It offers an integrated data model for both database and general purpose manipulation by providing database functionality and iterators to allow sets of objects to be manipulated as declaratively as database query languages based on relational calculus. It uses the language O++ based on, and upward compatible with, C++, to define, query and manipulate the database.

Ode is active in that it defines constraints and triggers associated with objects. Constraints and triggers in Ode have separate facilities since the two are logically independent. Constraints ensure the consistency of the database state but triggers have a more general applicability.

Updates that violate constraints are not allowed. A class can have constraints associated with it and all instances of that class must satisfy them. A derived class inherits all the constraints of its parent class and new constraints can be added.

Constraints in Ode can either be *hard* or *soft*. Hard constraints are checked at the boundaries of public member functions that update objects. Soft constraints are checked at the end of transactions. Hence, hard constraints can be violated only within the boundaries of member functions whereas soft constraints can be violated within a transaction. This distinction allows the constraint programmer to improve efficiency of constraint checking by deferring checks of less important constraints.

Constraints are specified in the class definition section as follows:

constraint:

$constraint_1$: $handler_1$;

$constraint_2$: $handler_2$;

...

$constraint_n$: $handler_n$;

where $constraint_i$ is a boolean expression involving components of the class involved and $handler_i$ is an action to be performed when the constraint is violated. The keyword *soft* precedes the keyword *constraint* for a soft constraint. If there is no handler then the violating transaction is simply aborted. The handler is used to manipulate the data so that the constraint is no longer violated. After the handler is executed the constraint is checked again and if violated the transaction is aborted.

Consider the following hard constraint:

constraint:

state == Name("NY") || Name("");

This forces the state to either be undefined or "NY". A transaction attempting to set the state to something else is aborted.

Triggers monitor the database for certain conditions that require the execution of an associated action. Triggers are specified in the class definition and consist of a condition and an action. Triggers apply only to the specific objects for which they were activated.

If a trigger is active and its condition becomes true then its action is executed in a separate transaction (unlike constraints). Triggers that fire will be executed only if the transaction causing them to fire commits successfully.

Ode supports two kinds of triggers: *once-only* (default) and *perpetual.* A once-only trigger is automatically deactivated after it has "fired", and it must be explicitly reactivated to fire again. Perpetual triggers are reactivated automatically after each firing. A trigger T_i associated with an object whose id is o-id is activated (reactivated) by the call:

o-id $\rightarrow$ T_i(arguments)

This activation returns a trigger id which can be used to manually deactivate the trigger:

deactivate(trigger_id)

The trigger syntax is:

trigger:

[perpetual] T_1 (parameter-decl$_1$) : trigger-body$_1$
[perpetual] T_2 (parameter-decl$_2$) : trigger-body$_2$
...
[perpetual] T_n (parameter-decl$_n$) : trigger-body$_n$

where T_i are the trigger names and the parameters can be used in trigger bodies, which are of the form:

trigger-condition => trigger-action

For example, consider the following trigger:

trigger:

order() : qty < reorder_level() ==> place_order();

This trigger will fire when *qty* becomes less than *reorder_level()*. The response will be to execute the function *place_order()*.

A second form of trigger body is used for specifying timed triggers. Once activated, a timed trigger must fire within the specifyed period or else the timeout-action is performed. These have the form:

within expression ? trigger-condition => trigger-action [:timeout-action]

Gehani *et al.* [20] specify complex events for Ode. These include the *relative*, *prior*, and *sequence* operators.

Postgres

Postgres is an extended relational DBMS and is the successor of the Ingres relational DBMS [36, 37, 38]. It extends Quel to the Postquel query language for data access. Postgres allows the definition of general rules that have a wide range of applications including view management, triggers, maintenance of integrity constraints, protection, referential integrity control and versioning.

Postgres rules have the following syntax:

ON event (TO) object WHERE
POSTQUEL-qualification
THEN DO [instead]
POSTQUEL-command(s)

where *event* is one of retrieve, replace, delete, append, new or old. The object is either the name of a class or class.column. The keyword *instead* is used to specify actions that are to be executed instead of the action which caused the rule to trigger. If instead is missing then the rule specifies actions that are to be taken in addition to the trigger causing actions. These actions are specified in the POSTQUEL-command(s) section of the rule declaration. These commands are Postquel commands with an enhanced syntax to allow reference to new or current states of a class.

Postgres rule control can be set to *forward* or *backward* chaining control mechanisms [38]. For example, suppose we wished to ensure that Joe's salary is always equal to Fred's. We could use the rule in Figure 2.1(a). Whenever an adjustment to Fred's salary occurs, Joe's salary will be updated automatically by the system (forward chaining). On the other hand, we could use the rule in Figure 2.1(b) to enforce the constraint. In this case every time Joe's salary is retrieved we retrieve Fred's salary instead (backward chaining) and Joe's salary is not explicitly stored.

These two different methods of rule control can be used to improve rule efficiency. If there are a high number of retrievals of Joe's salary and a low number of updates to Fred's salary then forward chaining would be most efficient. However, if there are a high number of updates to Fred's salary and a low number of retrievals of Joe's salary then backward chaining would be preferred.

Postgres has two approaches to rules. The first is through run-time record level

```
on new EMP.salary where
EMP.name = "Fred"
then do replace
E (salary = new.salary)
from E in EMP
where E.name = "Joe"
```

(a)

```
on retrieve to EMP.salary where
EMP.name = "Joe"
then do instead retrieve
(EMP.salary)
where EMP.name = "Fred"
```

(b)

Figure 2.1: Example Postgres rules

```
on replace to EMP.salary
then do
append to AUDIT
(name = current.name,
 salary = current.salary,
 new = new.salary, user = user())
```

Figure 2.2: Auditing rule

processing. This rule system is called when individual records are accessed. For example, in the rule in Figure 2.1(a) the record-level rule system places a marker (which contains this rule's identifier) on the salary attribute of Fred's instance. If the executor accesses a marked attribute then it will call the rule system to handle rule processing. After this is complete the executor will continue processing the original transaction.

The second approach is through a query rewrite module. This converts a user command to a more optimal alternate form. Consider the rule in Figure 2.2 and the incoming user command of Figure 2.3. This command will clearly cause the rule to fire once per employee over 50 years of age (a large overhead). The query rewrite rules system will rewrite the user command to the more efficient commands of Figure 2.4.

The record-level rule system performs well when there are a large number of small-scope rules whereas the query rewrite rules system works better if there are a small number of large-scope rules. It turns out that the two rules systems are complementary and Stonebraker and Kemnitz [38] explore a rule chooser which suggests the best implementation.

```
replace EMP
(salary = 1.1 * EMP.salary)
where EMP.age > 50
```

Figure 2.3: Incoming user command

```
append to AUDIT
(name = EMP.name, salary = EMP.salary,
new = 1.1 * EMP.salary, user = user())
where EMP.age > 50

replace EMP
(salary = 1.1 * EMP.salary)
where EMP.age > 50
```

Figure 2.4: Rewritten user command

In Postgres there are four different types of rule activation policies:

immediate - same transaction
immediate - different transaction
deferred - same transaction
deferred - different transaction

The rule to set Joe's salary to Fred's (Figure 2.1) must run immediately in the same transaction. If the original transaction is aborted then the rule is no longer applicable and does not need to update Joe's salary. However, the rule in Figure 2.2 must be activated immediately in a different transaction. If the original aborts we still wish for the auditing to occur.

The rule of Figure 2.1(a) sets Joe's salary to Fred's whenever Fred's is updated. The activation policy for this rule is *immediate - same transaction.* If the transaction which updates Fred's salary aborts then we want the trigger to abort (as Joe's salary no longer needs updating). The trigger action is part of the triggering transaction so we get the desired results. However, the rule of Figure 2.2 is activated immediately in a *different* transaction. If the triggering transaction aborts we still wish the auditing to occur.

Postgres rules have a large range of applicability. For example, take the following Postgres view:

define view TOY_EMP(EMP.all)
where EMP.dept = "toy"

This view is compiled into the following Postgres rule:

on retrieve to TOY-EMP

then do instead retrieve (EMP.all)
where EMP.dept = "toy"

A query retrieving tuples from the TOY-EMP relation will trigger this rule and the correct tuples from EMP will be retrieved instead.

Rules in Postgres can be grouped into *rulesets*. A ruleset is hierarchically structured and is defined as follows:

Define Ruleset ruleset_name
[inherits ruleset {,ruleset}]
[init_script proc-name]
[cleanup_script proc-name]

The *inherits* clause allows common collections of rules to be shared among multiple rulesets. The *init_script* and *cleanup_script* procedures each contain a script of commands to be run when the ruleset is activated or deactivated, respectively. A ruleset can be removed with the following command:

Remove Ruleset ruleset_name

Rulesets can be activated using the following command:

Activate ruleset_name
[i_script]
[late_signal]
[auto_deactivate]

When set *i_script* flag indicates that the initialization script is to be run. The *late_signal* flag indicates that the user wishes to be notified after no new inferences

have been made as a result of the ruleset activation. The *auto_deactivate* flag indicates that the ruleset is to be deactivated as soon as no new inferences have been made as a result of activation.

Rulesets can be deactivated using the command:

Deactivate ruleset_name
[d_script]

When set the *d_script* flag indicates that the deactivation script should be run.

Iris

Iris is an object-oriented DBMS developed at Hewlett-Packard Laboratories [29]. Iris can automatically notify application programs when stored or derived data changes. Iris uses database *monitors* to observe changes in the contents of database objects (e.g., the current price of some commodity or the location of some ship). Monitors can also observe changes to derived data (e.g., the highest paid employee in a department).

Application programs house tracking procedures callable by a database monitor if the value of some monitored data has changed. The DBMS does not transmit monitored data to tracking procedures but merely invokes them. The tracking procedures then retrieve any data required for its monitoring action and takes appropriate action.

Tracking procedure invocations can either be *local* or *external*. An invocation is local if the program containing the tracking procedure caused the change to the database which in turn caused the tracking procedure to be invoked. The invocation is external if it is the result of changes made by some other process. Local tracking

procedures are invoked before updates have been committed and external tracking procedures are invoked after the commit. Local tracking procedures are invoked synchronously (the system waits for them to return) and external ones are invoked asynchronously. There is also a no-interrupt option where the DBMS accumulates notifications until the process is ready to handle them.

Monitors are expensive so restricting them appropriately may be required. They can be *time localized*, where they are only active during a limited time period or *client localized* by deactivating them when there is no client that needs them. Monitors can be *object localized*, meaning that if the objects it involves are not being accessed then the monitor may be deactivated. Finally, they can be *property localized* so that just the attributes of interest are monitored.

The change detection algorithm used by monitors has four steps. First, every object that is updated by a transaction is tracked in a *virtual table*. Second, at monitor checking time each object in the virtual table is checked for monitors defined on them. Next, for each monitor detected object, a more expensive instance detection is done to test if its value changed. Finally, if the value changed, notifications are sent to the corresponding client processes.

Beeri and Milo Beeri *et al.* [7] describe a formal model for an active object-oriented database. In this model, method execution causes the triggering of actions. Methods serve as triggers when they are defined as:

$$\text{type method} = [\text{M-name,M-code,}\{\text{triggered-action}\}]$$

where *M-name* is the name of the method, *M-code* is the method code, and each element in *triggered-action* is the name of an action. Whenever the method is invoked by an object, the actions in the set are triggered for future execution based

on a programmer specified *execution interval*, within which the triggered actions must be executed. This interval consists of a start event and an end event:

type execution-interval = [start-event,end-event]

The triggered action must be executed as early as *start-event* and no later than *end-event*. Example events include:

- Method invocations,
- Points in time such as "12:00 am" or "Monday morning",
- Conjunctions of events using *and*, and
- *any time* which is used to specify intervals with only one end point.

Suppose we have a client with multiple account objects, each with a method: *update_account*. Further, assume we have a derived attribute called *account_balance* which is the sum of all the accounts and each of the *update_account* methods have a triggered action that updates *account_balance* when invoked. Let *UPDATE_ACCOUNTS* be a higher level method that invokes the *update_account* of each object to update each account to a new value. If we now execute *UPDATE_ACCOUNTS* then the *account_balance* must be recomputed every time an *update_account* method is called by *UPDATE_ACCOUNTS*, which introduces a very large and unnecessary overhead.

This overhead is avoided if the execution interval for the triggered action is defined to be [*UPDATE_ACCOUNTS*, *get_balance*], where *get_balance* reads *account_balance*. This indicates that the *account_balance* should be updated somewhere between the end of *update_accounts* and the start of *get_balance*. The scheduler will perform the update after *UPDATE_ACCOUNTS*, thereby avoiding the overhead.

There are three options in the case of triggering method failure. The triggered action could be rolled back (if it was executed) or deleted (if still pending), or the triggered action can be executed as planned. This is done by using *abort* or *ignore* statements in the triggered action description.

If a triggered action fails several options are possible. The action could be retried or the triggering method could be aborted using *ignore* and *abort(trans)*, respectively. Other options include: *try n times before aborting*, and *run t instead* where t is a compensating transaction.

The triggered action syntax is:

type triggered-action-description =
[triggered-action,
act-to-perform(location,parameters),
execution-interval,
scheduling-information,
triggered-action-fail,
trigger-fail]

Triggered-action is the trigger identifier and *act-to-perform(location,parameters)* is the action to perform at the object *location*. We have already discussed the execution-interval. *Scheduling-information* contains information such as priorities. The *triggered-action-fail* and *trigger-fail* fields indicate what to do if the triggered action aborts or triggering method fails, respectively.

Actions that are triggered may not be executed immediately so relevant information is stored about them until they are executed. The logical storage place for this information is within the object itself (assuming at this point that the action does not involve multiple objects).

Both an object's structure and methods are extended to handle triggered actions. Its structure is augmented with an *active* part that contain records of the form:

```
type active-action =
        [triggered-action(parameters),
        triggering-method-identity(location),...]
```

Records of this type each represent one action waiting for execution. Whenever a triggering method executes, its triggered action information is inserted into an *active_action* record and added to the object's active part.

An object's methods are extended to include methods to insert, retrieve, execute, and delete actions. These methods share the properties of regular methods in that they can be inherited and overridden.

To execute a method (m) a transaction called *global_control_m* consisting of five subtransactions is created. The first subtransaction *insert_start_m* inserts records, corresponding to those triggered actions of m that do not use any of the output parameters of m, into the active part of the object. The second subtransaction, *execute_start_m*, checks the contents of the active part of the object involved and executes triggered actions that must be executed before the start of m. Some triggered pending actions that do not need to be executed before the start of m can also be executed. The third subtransaction invokes m. If m calls any other methods, these are scheduled as subtransactions of m and are handled in the same manner. Next, *insert_end_m*, inserts records into the active part of the object corresponding to triggered actions that use output parameters of m. This is similar to *insert_start_m*. The final subtransaction, *execute_end_m*, executes the actions

that must be executed no later than the end of m, and possibly some others. It is similar to *execute_start_m*.

Quasi-copies A *Quasi-copy* is a cached value that can deviate from the original in a controlled way. The management of quasi-copies is called *quasi-caching*. These were proposed by Alonso *et al.* [3, 4] for use in large multidatabase systems to reduce the high communication costs associated with continually accessing data on remote database systems.

Quasi-caching differs from traditional caching because quasi-copies are not necessarily updated as soon as the original is changed. Since MDBSs have a high communication cost associated with updating these copies quasi-caching allows the user of the data to specify exactly what data is to be cached (*selection*) and how far it may deviate from the original (*coherency*). For example, the user may specify that a copy may not diverge by more than 10% from the original, or that the information may be no more than 1 hour old. The home site of this data will then send updates whenever these coherency conditions are violated.

Information flow within a large multidatabase system with quasi-copies is similar to the flow in many "real organizations". For example, the manager of a company is not told every time an employee is hired or leaves. The information is filtered so that only periodic information is passed along such as personnel changes or when an exceptional condition occurs (e.g., a mass hiring of employees). Hence, quasi-caching provides a very natural way of dealing with distributed data in very large multidatabase systems and also has a reasonable performance overhead [4].

Identity Connection

The *Identity connection* has been introduced for modeling the update propagation of replicated data within autonomous and distributed database systems [32]. An identity connection links copies of data that may be located at different sites and specifies the consistency requirements between them.

One of the copies specified in an identity connection is called the primary copy and the others are called secondary copies. A *temporal constraint* is specified with every identity connection to specify when an update on the primary copy must propagate to the secondary copies. To maintain a connection, whenever an update occurs to the primary copy a transaction containing the temporal constraint is submitted to the sites of the secondary copies. Transaction schedulers at secondary sites determine whether they can satisfy the temporal constraint using a *satisfiability test.* If a scheduler can satisfy the constraint, it will *promise* the primary site to execute the transaction in accordance with the temporal constraint. The update at the primary site can commit as soon as it receives promises from all secondary sites.

Barker Barker [6] defines a formal multidatabase system model. A multidatabase system consists of a number of autonomous local database systems each with their own DBMS and a multidatabase management system layer (MDMS) logically above them.

There are two types of transactions in this multidatabase system model: *local* ones that are submitted to each DBMS and *global* ones that are submitted to the MDMS. Local transactions execute on the database where they were submitted while global transactions may access multiple databases. Global transactions are managed by the MDMS which parses them into *global subtransactions* that are submitted to the local DBMSs for execution. The DBMSs are responsible for

executing local transactions and global subtransactions concurrently and for local reliability. The synchronization of global transactions is the responsibility of the MDMS.

Barker defines MDB-Serializability as a correctness criterion for multidatabase systems and presents concurrency control algorithms for global transactions which ensure this correctness. This thesis proposes to add active functionality to Barker's model and so it is not discussed to any great length here. Instead, relevant portions of the model are presented throughout this thesis as we modify it to support active behavior accordingly.

Chapter 3

A Formal Model of AMDS

This chapter presents the basic definitions necessary to describe an active multidatabase system. This chapter is presented as follows: Section 3.1 defines the various transaction types that are part of this model. Section 3.2 presents the definitions for *global triggers* in this environment. The definitions of various types of histories are given in Section 3.3 and *transaction families* are defined in Section 3.4. A new form of serializability called active multidatabase serializability is described in Section 3.5. *Active multidatabase serializability graphs* are described in Section 3.6 and used to reason about a histories correctness with the AMDB serializability theorem presented in Section 3.7. Finally, Section 3.8 summarizes the model by explicitly stating the assumptions imposed by this model.

3.1 Transaction Types

An active multidatabase system contains two types of transactions: *local* and *global*. Local transactions are submitted to each DBMS and global ones are submitted to

the AMDMS (Figure 1.2). Local transactions execute on a single database whereas global transactions may access multiple databases.

An AMDS also contains *global triggers* which can be fired by executing global transactions. Before formally defining global triggers we present definitions from Barker [6] that will be used as the foundation for our model.

Definition 3.1.1 *(Local Database):* Each of the autonomous databases that make up a multidatabase is called a *local database* (DBMS). The set of data items stored at a DBMS, say i, is denoted $\mathcal{LDB}^i$. The set of all data in the multidatabase can be defined as $\mathcal{MDB} = \bigcup_i \mathcal{LDB}^i$. ■

Definition 3.1.2 *(Local transaction):* A transaction T_i submitted to DBMS j (denoted $DBMS^j$) is a *local transaction* (denoted LT_i^j) on $DBMS^j$ if $OS_i \subseteq \mathcal{LDB}^j$. ■

Definition 3.1.3 *(Global transaction):* A transaction is a *global transaction* (GT_i) iff:

1. $\neg\exists LDB^j$ such that $\mathcal{BS}_i \subseteq \mathcal{LDB}^j$ or
2. GT_i is submitted to $DBMS^k$ but $\mathcal{BS}_i \subset \mathcal{LDB}^r$ $(k \neq r)$. ■

Item (1) states that global transactions, submitted to the AMDMS, access data items stored in more than one database. Item (2) represents the case where a user working on one DBMS requires access to the data stored and managed by another.

Global transactions do not directly access databases. They are parsed into a set of *global subtransactions* that are submitted to the local DBMSs. Thus, the subtransactions work on behalf of the global transaction. Our definition of a global

subtransaction has additional information supporting the detection of global trigger events. Global subtransactions are defined in terms of the data items referenced (i.e., their base-set), the global transaction creating them, and the *global triggers* they could fire.

Definition 3.1.4 *(Global subtransaction):* A global subtransaction submitted to $DBMS^j$ for global transaction GT_i (denoted GST_i^j) is a transaction where:

1. $\Sigma_i^j \subseteq \Sigma_i$ and
2. $\mathcal{BS}_i^j \subseteq \mathcal{LDB}^j$, where $\mathcal{BS}_i^j$ is the base-set of GST_i^j. ■

3.2 Global Triggers

Global triggers are fired when global subtransactions cause events to occur. A global trigger is executed similar to a global transaction however it has additional information, called the triggers *coupling mode*, that describes how it is executed.

Definition 3.2.1 *(Global trigger):* A global trigger is an ordered triple $GTR_k = (GT, E, M)$ where:

1. GT is the trigger action (which is in the form of a global transaction),
2. E is some database event, and
3. M is the triggers *coupling mode.* ■

Global transactions can cause the execution of global triggers so we introduce a new operation: *fire*$(GT_{j.n}$*,cause?)*. If one or more of the operations of a global

transaction cause or may cause the event of some global trigger GTR_j to occur, then $fire(GT_{j.n}, cause?) \epsilon OS_i$. The value of *cause?* is true if and only if the trigger is *causally dependent* (see Section 3.2.2)

If a global subtransaction GST_j^i causes a global trigger GTR_k to fire (i.e. causes the event of GTR_k to occur) then GST_j^i executes its $fire(GT_t, cause?)$ operation. This causes the submission of a global transaction GT_t which will execute the action of GTR_k. Note that $t = k.n$ and n is the copy identifier. For example, the third firing of GTR_k means $n = 3$. Thus, all GTs executing on the AMDS are still uniquely identifiable by their subscripts.

3.2.1 Trigger Events

Global trigger events can occur because of a read or write to some data item or a complex function of these operations. The following definitions capture this intuition:

Definition 3.2.2 *(Simple Event):* An event is *simple* if it is a read or write operation. ■

Definition 3.2.3 *(Complex Event):* A *complex event* is composed of other events (which may be simple or complex). These include:

Disj(E_1, E_2)*:* occurs when either event E_1 or event E_2 occurs.

Conj(E_1, E_2)*:* occurs when both events E_1 and E_2 have occurred, regardless of the order of occurrence.

Seq(E_1, E_2)*:* occurs when both events E_1 and E_2 have occurred but E_1s occurrence precedes E_2.

Clos(E_1, N): occurs when the event E_1 has occurred N times. ■

Some examples of complex events include:

(a) Disj(r(x),w(x)): Occurs when the data item x is *accessed.*

(b) Conj(Disj(r(y),w(x)),w(z)): Occurs when either $r(y)$ or $w(x)$ have occurred and $w(z)$ has occurred.

(c) Seq(r(x),w(y)): Occurs when y is written and x has already been read by the same global transaction.

(d) Conj(Conj(r(a),r(b)),Disj(r(c),r(d))): Occurs when a, b, and either of c and d have been read by some global transaction.

Complex events have a natural nested structure so they are described by a tree. The leaf nodes of a complex event tree are always simple and are referred to as the event's *simple events.* Figure 3.1 shows the event trees for the example complex events above.

3.2.2 Coupling Modes

The coupling mode M specifies how the triggered transaction is executed with respect to the triggering transaction. It describes various dependencies between the triggering transaction and triggered transaction. The modes we support include the *independent* (Idep), *causally dependent*(Cdep), *trigger dependent*(Tdep), and *fully dependent*(Fdep) modes:

- **Idep (independent):** The global trigger is executed and completes independent of the triggering transaction so the trigger commits or aborts independent of the termination decision of the triggering transaction. Since

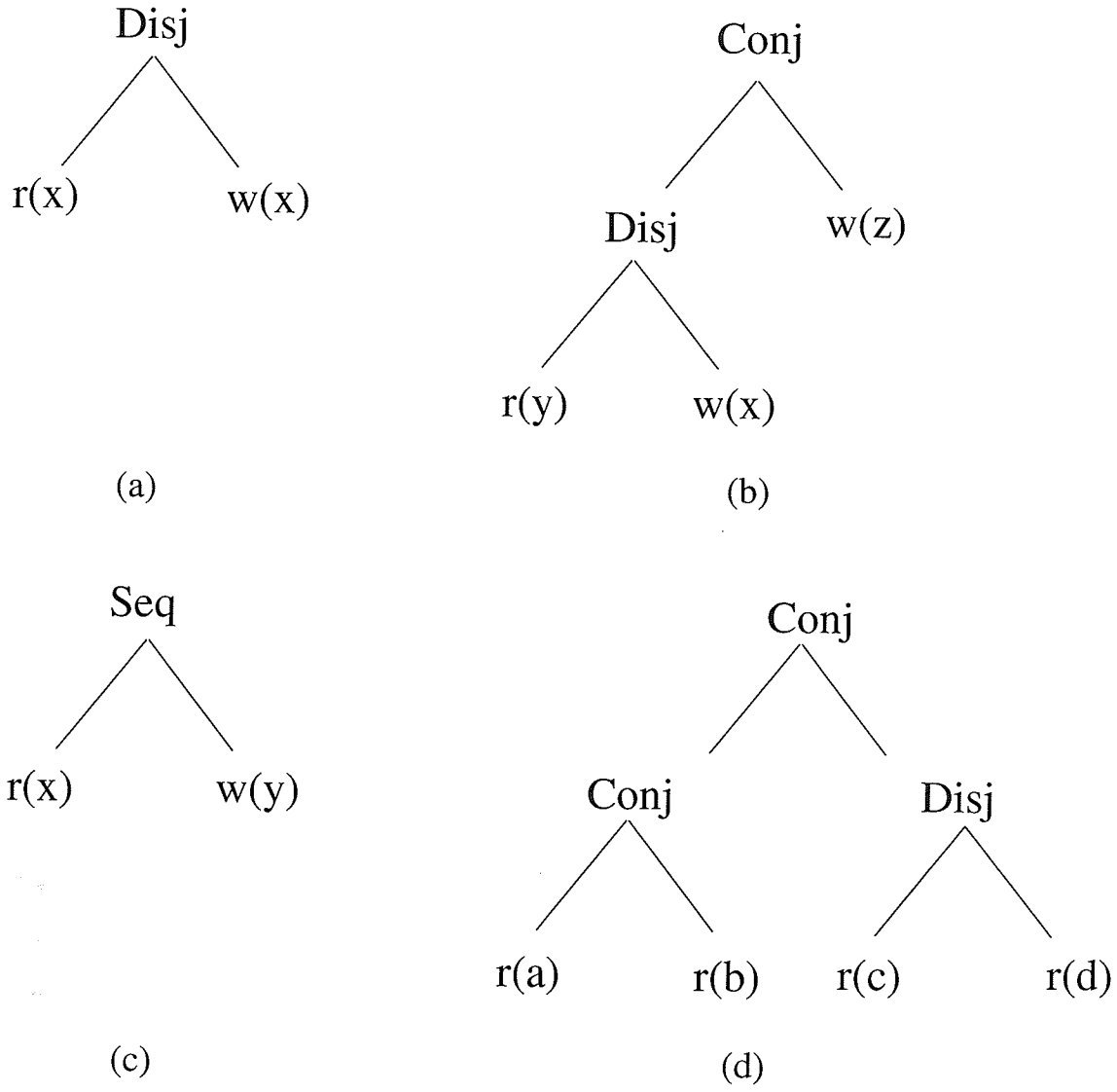

Figure 3.1: Sample Event trees

these are independent transactions the serialization order of the transactions is irrelevant.

- **Cdep (causally dependent):** The trigger makes its termination decision independent of the triggering transaction however, the trigger is *causally dependent* on the triggering transaction so the triggering transaction must not "see" any of the triggers results. Therefore, the trigger must be serialized after the triggering transaction in the resulting history.

- **Tdep (trigger dependent):** The trigger is *commit dependent* on the triggering transaction so the trigger must abort if the triggering transaction aborts. The trigger is also causally dependent.

- **Fdep (fully dependent):** The trigger and triggering transaction are commit dependent on each other. The trigger is causally dependent. A trigger of this mode is considered a *child* of the triggering transaction (see Section 3.4).

An airline reservation system is used to illustrate these modes composed of several airline DBMSs[1]. Any one database stores information regarding flights, passengers, security, etc.

- **Idep:** Our reservation system requires a global trigger to log writes to airplane data items at local database $DBMS^i$. Whenever a transaction writes to an airplane data item at $DBMS^i$, a global trigger to update the security access logs situated at local database $DBMS^j$ must fire. The global trigger is not commit dependent on the triggering transaction because we do not want the

[1]This is intended to illustrate the desired effects, the details of how each will be supported in an AMDS is deferred to later sections.

trigger to abort if the triggering transaction aborts. All accesses, committed or not, must be logged for security purposes.

- **Cdep:** With the preceding trigger it was possible for the triggering transaction to be serialized after the triggered transaction. This is effectively like the security logging transaction executed before the transaction which made the update. If this is of concern to the users of the database then the causally dependent mode should be used. This will ensure that all results of the the security appear to have occurred after the triggering transaction. The logging transaction must commit even if the triggering transaction aborts.

- **Tdep:** Updates to the number of passengers on planes at any database fire a global trigger. The trigger checks the amount that the number of passengers changed. If the change is significant then the trigger will update a list of aggregate values at $DBMS^i$. If the deviation is insignificant then it will simply abort. The triggering transaction may still commit. If the triggering transaction decides to abort then the triggered action is no longer needed and must abort.

- **Fdep:** The system must support the update of replicated data. $DBMS^i$ and $DBMS^j$ have decided to replicate important flight information from $DBMS^i$ at $DBMS^j$. Whenever, a global transaction updates the flight information, a global trigger must fire to update the replicated information at $DBMS^j$. If the original aborts then the trigger should abort as well because the replicated data no longer needs to be updated. If the trigger aborts the original must abort so we do not have inconsistent data.

Figure 3.2 summarizes the various coupling modes with respect to commit and serializability restrictions.

Mode	Commit	Serializability
Independent	ϕ	ϕ
Causally Dependent	ϕ	$P_c \wedge C_c \vdash P \prec C$
Trigger Dependent	$P_a \vdash C_a$	$P_c \wedge C_c \vdash P \prec C$
Fully Dependent	$P_x \vdash C_x$ where $x \epsilon (a, c)$	$P_c \wedge C_c \vdash P \prec C$

Figure 3.2: Summary of Coupling Mode Implications

3.3 Histories

Histories are used to define correct executions within database systems. Barker's history definitions are adapted [6].

Definition 3.3.1 *(Local History):* Given a $DBMS^k$ with a set of local transactions $\mathcal{LT}^k$ and a set of global subtransactions $\mathcal{GST}^k$, a *local history* (LH^k) is a partial order $LH^k = (\Sigma^k, \prec^k_{LH})$ where:

1. $\Sigma^k = \bigcup_j \Sigma^k_j$ where Σ^k_j is the domain of transaction $T_j \epsilon \mathcal{LT}^k \bigcup \mathcal{GST}^k$ at $DBMS^k$,
2. $\prec^k_{LH} \supseteq \bigcup_j \prec^k_j$ where $\prec^k_j$ is the ordering relation for transaction T_j at $DBMS^k$, and
3. for any two conflicting operations $p, q \epsilon LH^k$, either $p \prec^k_{LH} q$ or $q \prec^k_{LH} p$. ■

This captures the ordering of transactions submitted to a particular database. Next we project the orderings of the global subtransactions submitted.

Definition 3.3.2 *(Global subtransaction history):* The global subtransaction history of a $DBMS$, say k, is defined by the partial order $GSH^k = (\Sigma^k_{GSH}, \prec^k_{GSH})$ where:

1. $\Sigma^k_{GSH} = \bigcup_j \Sigma^k_j$, where Σ^k_j is the domain of transaction $T_i \epsilon \mathcal{GST}^k$ and

2. $\prec^k_{GSH} \subseteq \prec^k_{LH}$. ■

A global history is the collection of all the global subtransaction histories of an AMDS.

Definition 3.3.3 *(Global History):* A global history $GH = (\Sigma_{GH}, \prec_{GH})$ is the union of all global subtransaction histories:

1. $\Sigma_{GH} = \bigcup_k \Sigma^k_{GSH}$,

2. $\prec_{GH} \supseteq \bigcup_k \prec^k_{GSH}$, and

3. for any two conflicting operations $p, q \epsilon GH$, either $p \prec_{GH} q$ or $q \prec_{GH} p$. ■

The final history definition describes all transaction executions on the AMDS. A global history is the combination of the local histories and the global history.

Definition 3.3.4 *(MDB History):* A multidatabase history (denoted MH) consists of n local histories and a global history (GH) is a tuple $MH =< \mathcal{LH}, GH >$ where $\mathcal{LH} = \{LH^1, LH^2, ..., LH^n\}$. ■

The following example is used throughout the thesis to illustrate important concepts. It extends the example presented in Barker [6].

Example 3.3.1 Two local databases constitute our example multidatabase system whose contents are: $\mathcal{LDB}^1 = d, e, f, g$ and $\mathcal{LDB}^2 = s, t, u, v$. Two global transactions are posed to the multidatabase as follows:

$$GT_1 : r_1(d); r_1(e); fire_1(GT_3, true); w_1(s); w_1(d); c_1;$$
$$GT_2 : r_2(d); r_2(u); w_2(s); w_2(d); c_2;$$

These generate the following global subtransactions:

$$GST_1^1 : r_1^1(d); r_1^1(e); fire_1^1(GT_3, true); w_1^1(d); c_1^1;$$
$$GST_1^2 : w_1^2(s); c_1^2;$$
$$GST_2^1 : r_2^1(d); r_2^1(d); c_2^1;$$
$$GST_2^2 : r_2^2(u); w_2^2(s); c_2^2;$$

Next, we define one global trigger on the multidatabase:

$$GTR_1 : ([r(g); r(s); w(t);], r(e), Tdep)$$

The action of GTR_1 is $[r(g); r(s); w(t);]$ and is executed when the event $r(e)$ occurs. This triggers coupling mode is $Tdep$ (trigger dependent). The firing of this trigger generates the following global transaction:

$$GT_{1.1} : r_{1.1}(g); r_{1.1}(s); w_{1.1}(t); c_{1.1};$$

The subtransactions for this global transaction are:

$$GST_{1.1}^1 : r_3^1(g); c_3^1;$$
$$GST_{1.1}^2 : r_3^2(s); w_3^2(t); c_3^2;$$

Further, we introduce local transactions into each DBMS as follows:

$$LT_1^1 : \hat{r}_1^1(e); \hat{w}_1^1(e); \hat{w}_1^1(d); \hat{c}_1^1;$$
$$LT_1^2 : \hat{r}_1^2(u); \hat{w}_1^2(u); \hat{c}_1^2;$$

The $\hat{r}(\hat{w})$ notation distinguishes local transaction operations from operations of global subtransactions. This is for notational convenience only.

Assume that the following local histories are produced at each site:

$$LH^1 : r_1^1(d); r_1^1(e); fire_1^1(GT_{1.1}, true); w_1^1(d); r_{1.1}^1(g); \hat{r}_1^1(e); r_2^1(d); \hat{w}_1^1(e); \hat{w}_1^1(d); w_2^1(d); c_1^1; c_2^1; c_{1.1}^1; \hat{c}_1^1;$$
$$LH^2 : r_1^2(u); r_{1.1}^2(s); w_{1.1}^2(t); w_1^2(s); w_2^2(s); \hat{r}_1^2(u); \hat{w}_1^2(u); c_1^2; c_2^2; c_{1.1}^2; \hat{c}_1^2;$$

The following global subtransaction histories can be projected:

$$GSH^1 : r_1^1(d); r_1^1(e); fire_1^1(GT_{1.1}, true); w_1^1(d); r_{1.1}^1(g); r_2^1(d); w_2^1(d); c_1^1; c_2^1; c_{1.1}^1;$$
$$GSH^2 : r_1^2(u); r_{1.1}^2(s); w_{1.1}^2(t); w_1^2(s); w_2^2(s); c_1^2; c_2^2; c_{1.1}^2;$$

The global history is the partial order which combines GST^1 and GST^2 as $GH = \{GSH^1 \cup GSH^2\}$ and the multidatabase history is $MH =< \{LH^1, LH^2\}, GH >$. ■

3.4 Global Transaction Families

Recall, the full dependency coupling mode defines a close relationship between the triggering and triggered transactions but independent transactions (either fired by a trigger or submitted by a user) are themselves the top-level of their own tree. This notation is formalized by:

Definition 3.4.1 *(Top-Level GT):* A Top-Level GT_i is one which is submitted by a user or has an independent, causally dependent, or trigger dependent coupling mode. That is, the only GTs which are not top-level are ones with a full dependency mode. ■

Definition 3.4.2 *(Global transaction family):* A *global transaction family* GTF_i is a set of global transactions containing:

1. GT_i, a top-level global transaction, and
2. $\{GT_j \mid GT_j$ is a full global transaction fired by GT_i or fired by a descendent of $GT_i\}$ ■

We use an extended notation for global transactions and subtransactions to indicate their transaction family. If GT_i is a global transaction, GST_i^j is a subtransaction of GT_i, and GT_i belongs to GTF_k, then we may refer to GT_i as $GT_{i(k)}$ and GST_i^j as $GST_{i(k)}^j$. This extended notation explicitly states the GTF to which a particular GT or GST belongs. This is illustrated by the following example.

Example 3.4.1 The following global transaction is submitted to the AMDMS:

$$GT_1 : r_1(a); fire_1(GT_{5.1}, true); w_1(b); fire_1(GT_{1.1}, true); w_1(a); c_1;$$

The execution of GT_1 causes the execution of five global triggers:

$$GTR_1 : ([r(c); w(i); fire(GT_{2.1}, true); w(d); w(j); fire(GT_{4.1}, false);], w(b), Fdep)$$
$$GTR_2 : ([w(x); w(y);], w(i), Tdep)$$
$$GTR_3 : ([w(n);], w(d), Fdep)$$
$$GTR_4 : ([r(m); w(m);], w(j), Idep)$$
$$GTR_5 : ([w(z);], r(a), Fdep)$$

Global transactions $GT_{1.1}$, $GT_{2.1}$, $GT_{3.1}$, $GT_{4.1}$, and $GT_{5.1}$ execute the action components of GTR_1, GTR_2, GTR_3, GTR_4, GTR_5 respectively. GT_1 fires two full dependency triggers GTR_1 and GTR_5. $GT_{1.1}$ fires $GT_{2.1}$ (commit dependency),

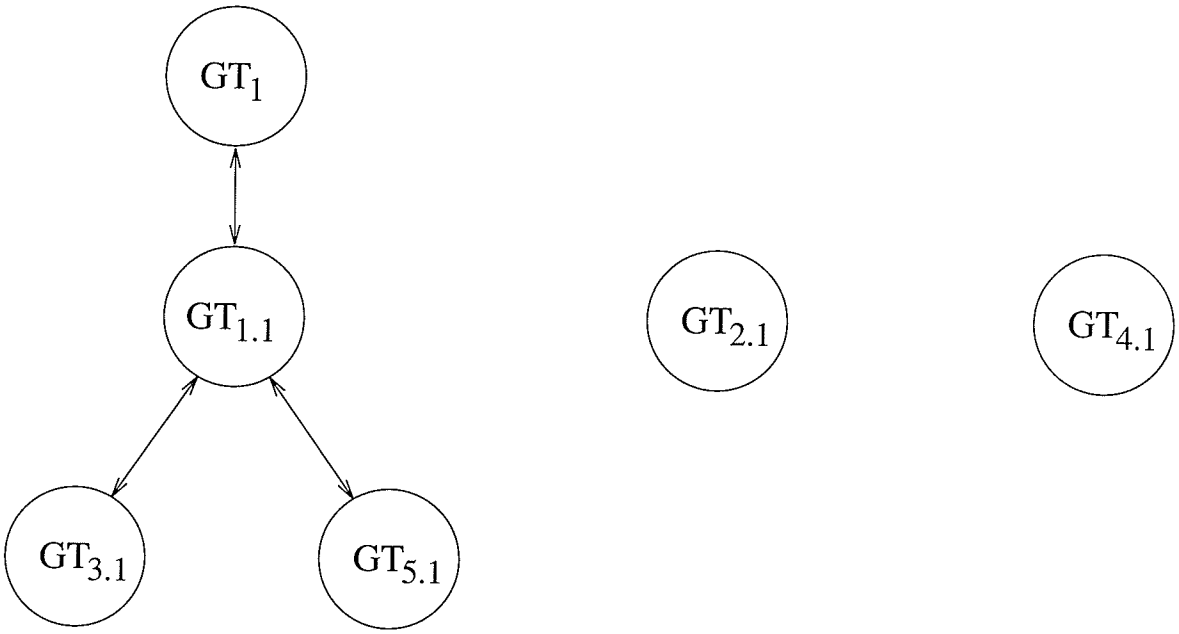

Figure 3.3: Depiction of Transaction Families for Example 3.4.1

$GT_{3.1}$ (full dependency), and $GT_{4.1}$ (independent). GT_1, $GT_{1.1}$, $GT_{5.1}$, and $GT_{3.1}$ are all members of GTF_1 and $GT_{2.1}$ and $GT_{4.1}$ form their own families. Figure 3.3 depicts the families. ■

The computational model discussed thus far is depicted in Figure 3.4. This figure extends the simple architecture of Figure 1.1 to include transactions and trigger firings.

3.5 AMDB-Serializability

The multidatabase serializability correctness criterion is insufficient for the AMDB environment because it does not support *causality*. For example, if one global transaction causes another and the second is causally dependent on the first then

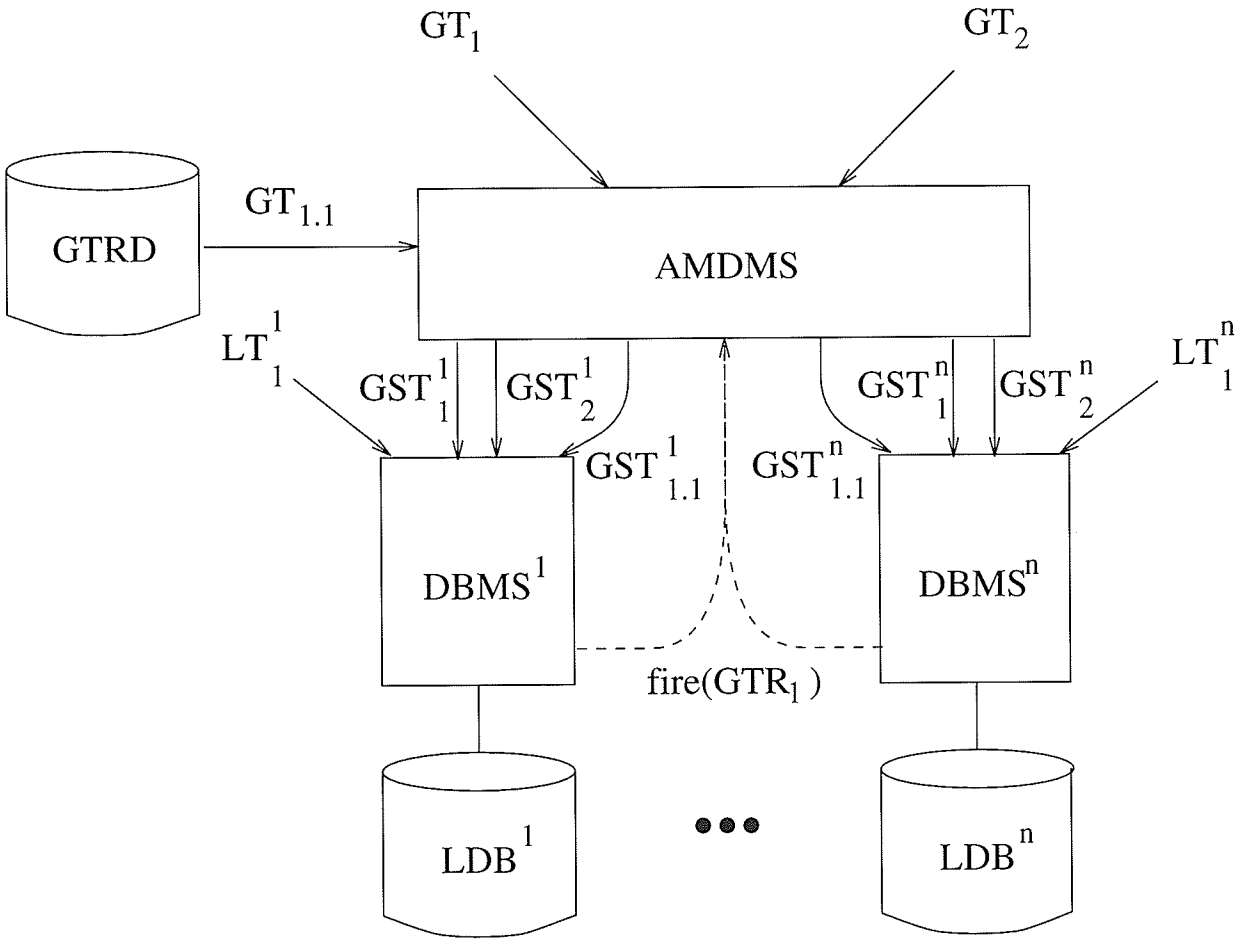

Figure 3.4: Depiction of the Computational Model

the first must not see the results of the second. Thus, the original transaction must be serialized before the triggered transaction in the multidatabase history.

Furthermore, MDB-serializability does not capture global transaction family orderings. Subtrees of transaction families must execute as isolated units. That is, it must appear as if all of the global transactions of a subtree have executed without some other transaction inbetween. If a transaction triggers one or more subtrees then the effects of the triggered subtrees must occur immediately after the parent. This prevents another transaction from reading from the parent and making decisions based on data that was to be "corrected" by a triggered subtree. The full dependency coupling mode tightly integrates a subtree, and thus a family, together as an atomic, isolated unit.

The following definitions present a suitable correctness criterion.

Definition 3.5.1 *(Causal dependency graph):* Given an arbitrary active multidatabase history (MH), its causal dependency graph $CDG(MH) = (\Gamma, \gamma)$ is:

1. Γ a set of labeled vertices representing global transactions.
2. γ is a set of arcs connecting two vertices in Γ. A γ-arc is formed from GT_j to GT_i if $fire_i(GT_j, true) \epsilon MH$. ■

Consider the following example.

Example 3.5.1 Suppose we have an active multidatabase history MH. The GH portion of MH is:

$$GH = \{\{r_1^1(a); fire_1^1(GT_{5.1}, true); w_1^1(b); fire_1^1(GT_{1.1}, true); r_{1.1}^1(c); w_{1.1}^1(d); \\ fire_{1.1}^1(GT_{3.1}, true); w_1^1(a); c_1^1; c_{1.1}^1; \}$$

$$\bigcup \{w^2_{1.1}(i); fire^2_{1.1}(GT_{2.1}, true); w^2_{1.1}(j); fire^2_{1.1}(GT_{4.1}, false); r^2_{4.1}(m);$$
$$w^2_{3.1}(n); w^2_{4.1}(m); c^2_{1.1}; c^2_{3.1}; c^2_{4.1}; \}$$
$$\bigcup \{w^3_{2.1}(x); w^3_{5.1}(z); w^3_{2.1}(y); c^3_{2.1}; c^3_{5.1}; \}\}$$

The CDG for MH is illustrated in Figure 3.5. Five firings exist in the history:

GT_1 fires $GT_{1.1}$

GT_1 fires $GT_{5.1}$

$GT_{1.1}$ fires $GT_{2.1}$

$GT_{1.1}$ fires $GT_{3.1}$

$GT_{1.1}$ fires $GT_{4.1}$

All but the last involve causal dependencies so there are four arcs in the CDG for the history. $GT_{1.1}$ is causally dependent on GT_1, $GT_{5.1}$ is dependent on GT_1, $GT_{2.1}$ is dependent on $GT_{1.1}$, and $GT_{3.1}$ is dependent on $GT_{1.1}$. ■

Note that causal dependencies are transitive. For instance, Example 3.5.1 $GT_{3.1}$ is causally dependent on $GT_{1.1}$ and GT_1.

Definition 3.5.2 *(AMDB-Serial):* A multidatabase history is AMDB-Serial iff:

1. every $LH \epsilon \mathcal{LH}$ is (conflict) serializable,
2. given a $GH = \{GST^n_1, ..., GST^m_r\}$, if $\exists p \epsilon GST^k_i, q \epsilon GST^k_j$ such that $p \prec_{GH} q$, then $\forall t, \forall r \epsilon GST^t_i, \forall s \epsilon GST^t_j, r \prec_{GH} s$, and
3. if $\exists GT_i$ and $GT_j \epsilon MH$ and a path exists from GT_j to GT_i in CDG(MH), then $\forall k, \forall r \epsilon GST^k_i, \forall s \epsilon GST^k_j, r \prec_{GH} s$.

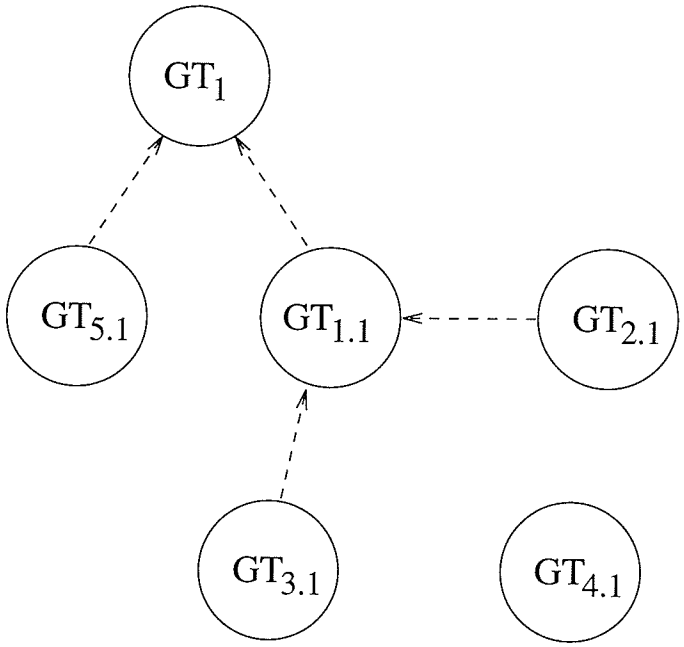

Figure 3.5: CDG for Example 3.5.1

4. if $\exists GT_i$ and $GT_j \epsilon MH$, GT_j a child of GT_i and $\exists p$ with $t \prec_{GH} p$ and $p \prec_{GH} s$ for some $t \epsilon GT_i$, $s \epsilon GT_j$ then $p \epsilon GT_k$, $GT_k \epsilon MH$, a descendent of GT_i. ■

The first condition states that all local histories are conflict serializable. The second condition states that if an operation of a global transaction precedes an operation of another global transaction at the same site, then all operations of the first transaction must precede all operations of the second transaction, at all sites. The first and second conditions together ensure that the multidatabase history is MDB-Serial. The third condition states that if one global transaction is causally dependent on another global transaction, either directly or indirectly, then all operations of the first must precede any operations of the second, at all sites. The fourth condition states that if an operation exists between an operation of some parent and an operation of a child then that operation must belong to a descendent of the parent.

This ensures that global transaction family orderings are correct.

Definition 3.5.3 *(Equivalence of Histories (≡)):* Two histories are conflict equivalent if they are defined over the same set of transactions and conflicting operations of nonaborted transactions are ordered in the same way. ■

Definition 3.5.4 *(AMDB-Serializable (AMDBSR)):* A MH is AMDB-serializable iff it is equivalent to an AMDB-Serial history. ■

3.6 Active Multidatabase Serializability Graphs

To reason about the AMDB-serializability of active multidatabase histories, we use a variation of the multidatabase serializability graphs (MSGs) defined by Barker [6]. MSGs alone are inadequate because they do not capture conditions 3 and 4 of Definition 3.5.2 (recall Example 3.3.1).

Barker's MSGs use $\lambda - arcs$ to represent the serialization order of transactions executing at the local databases and $\gamma - arcs$ to represent orderings of global transactions [6]. The MSG for this example is presented in Figure 3.6. Double headed arrows are used to represent $\gamma - arcs$ and single headed arrows for $\lambda - arcs$ Note that the graph is acyclic and by Theorem 4.1, page 41 in Barker [6] the multidatabase history is MDB-serializable. However, examining the GH portion of the multidatabase history tuple:

$$GH = \{\{r_1^1(d); r_1^1(e); fire_1^1(GT_{1.1}, true); w_1^1(d); r_{1.1}^1(g); r_2^1(d); w_2^1(d); c_1^1; c_2^1; c_{1.1}^1; \} \cup \{r_1^2(u); r_{1.1}^2(s); w_{1.1}^2(t); w_1^2(s); w_2^2(s); c_1^2; c_2^2; c_{1.1}^2; \}\}$$

we see that $r_{1.1}^2(s) \prec_{GH} w_1^2(s)$ at $DBMS^1$. Since $fire_1^1(GT_{1.1}, true) \epsilon$ GH this multidatabase history is not AMDB-serializable.

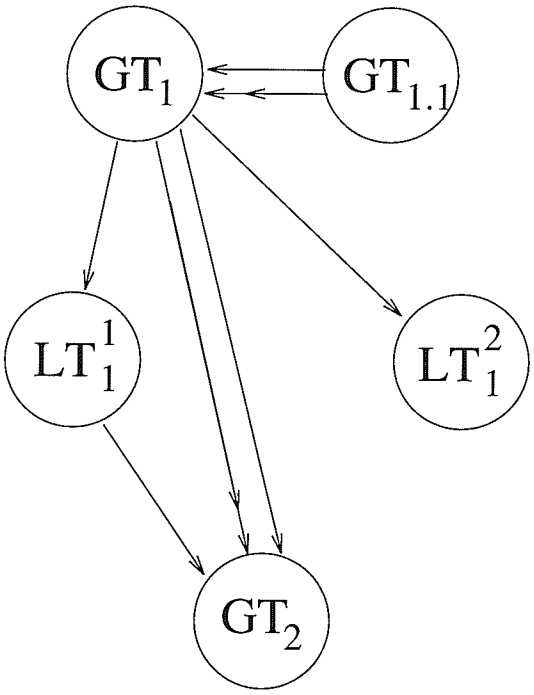

Figure 3.6: MSG for Example 3.3.1

We modify the serializability graphs presented in Barker to capture these violations.

Definition 3.6.1 *(Active Multidatabase Serializability Graph):* Given an arbitrary active multidatabase history (MH), its active multidatabase serializability graph is a digraph defined with the ordered five: $AMSG(MH) = (\Gamma, \Lambda, \gamma, \lambda, \pi)$. Each element of the ordered five is defined as follows:

1. Γ is a set of labeled vertices representing global transactions.

2. Λ is a set of labeled vertices representing local transactions.

3. γ is a set of arcs, each connecting two vertices in Γ. A $\gamma - arc$ is formed when two global transactions $(GT_i, GT_j \epsilon \mathcal{GT})$ conflict, if an operation of any GST_i^k precedes a conflicting operation of GST_j^k in MG, a $\gamma - arc$ is formed form the corresponding nodes in Γ from GT_i to GT_j.

4. λ is a set of arcs, each connecting two vertices in $\Gamma \bigcup \Lambda$ when two conflicting transactions T_i^k and T_j^k submitted to $DBMS^k$ so that T_i^k precedes T_j^k:

   (a) if $T_i^k, T_j^k \epsilon \mathcal{LT}^k$ a $\lambda - arc$ is formed from T_i^k to T_j^k.

   (b) if $T_i^k, T_j^k \epsilon \mathcal{GST}^k$ a $\lambda - arc$ is formed between GT_i and GT_j, respectively, i.e. $GT_i \rightarrow GT_j$.

   (c) if $T_i^k \epsilon \mathcal{GST}^k$, $T_j^k \epsilon\ \mathcal{LT}^k$ (or vice versa) a $\lambda - arc$ is formed from GT_i to LT_j^k (or reverse: $LT_j^k \rightarrow GT_i$).

5. π is a set of arcs each connecting two vertices in Γ. A $\pi - arc$ is formed from GT_i to GT_j whenever $fire_i(GT_j, true) \epsilon$ MH.

■

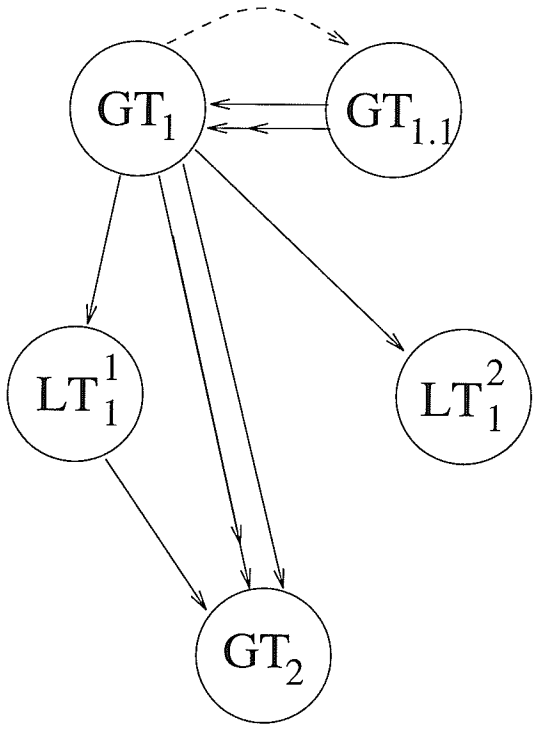

Figure 3.7: AMSG for Example 3.6.1

The first four elements of Definition3.6.1 are Barker's multidatabase serializability graph [6]. The fifth element is a set of arcs representing *causality.* For example, if a global transaction causes another global transaction and there is a causal dependency between them, then a $\pi - arc$ is formed from the original to the triggered. The significance of these arcs is described later.

Example 3.6.1 Recall Example 3.3.1. A history was described that was MDB-serializable but not AMDBSR. The AMSG for this example is illustrated in Figure 3.7. The graph is the same as the one in Figure 3.6 except it captures causal dependencies. There is a $\pi - arc$ from GT_1 to $GT_{1.1}$. ■

3.7 Active Multidatabase Serializability Theorem

AMSGs can be used to demonstrate when an arbitrary active multidatabase history is AMDB-serializable. Before presenting the serializability theorem we present definitions which will aid in the understanding of the theorem and its proof.

Definition 3.7.1 *($\pi-path$):* An AMSG(MH) contains a $\pi-path$ from GT_i to GT_j if there exists a path consisting solely of π arcs from GT_i to GT_j in AMSG(MH). ■

Definition 3.7.2 *($\lambda, \gamma - path$):* An AMSG(MH) contains a $\lambda, \gamma - path$ from GT_i to GT_j if only λ and γ arcs occur from GT_i to GT_j in AMSG(MH). ■

Definition 3.7.3 *(Causal Discrepancy):* An AMSG contains a causal discrepancy if $\exists$ a $\pi - path$ from GT_i to GT_j and $\exists$ a $\lambda, \gamma - path$ from GT_j to GT_i. ■

In other words, a causal discrepancy exists if there exists a GT_j which is causally dependent on GT_i and there is a path consisting of λ and γ arcs from $GT_j \rightarrow GT_i$ in the AMSG, then, assuming that the multidatabase history is MDBSR, GT_j will be ordered before GT_i. This is a violation of the causal dependency.

Definition 3.7.4 *(Family Order Discrepancy):* An AMSG(MH) contains a family order discrepancy if there exists a $\lambda, \gamma - path$ from GT_i to a child GT_j and there exists a node on this path (other than GT_i) which is not a descendent of GT_i. ■

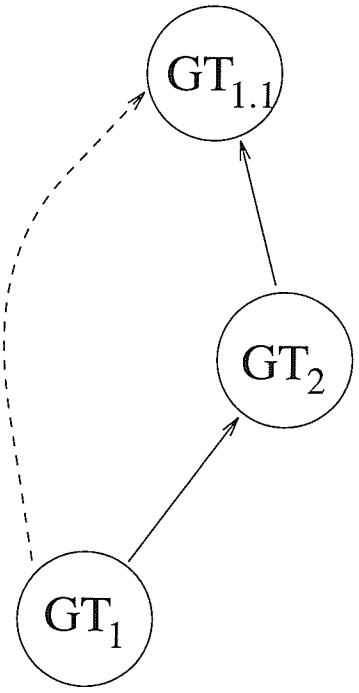

Figure 3.8: AMSG for Example 3.7.1

Example 3.7.1 Consider the AMSG of Figure 3.8. The graph shows that $GT_1 \prec GT_2 \prec GT_{1.1}$ which violates the family ordering described in item 4 of Definition 3.5.2. The path from GT_1 to its child $GT_{1.1}$ contains GT_2 which is not a descendent of GT_1. ■

We are now in a position to present the theorem.

Theorem 3.7.1 *(AMDB Serializability Theorem)* A multidatabase history (MH) is AMDB-serializable if and only if AMSG(MH) is $\gamma - acyclic$, $\lambda - acyclic$, does not contain any causal discrepancies, and does not contain any family order discrepancies.

Proof:

(if): Given a $\gamma - acyclic$, $\lambda - acyclic$, *causal discrepancy-free*, and *family order*

discrepancy-free AMSG for a multidatabase history MH; MH is AMDB-serializable. Without loss of generality, assume that $MH =< \mathcal{LH}, GH >$ refers to the committed projection of a multidatabase history.[2] Suppose the global history GH is defined over the set of transactions $\mathcal{GT} = \{GT_1, ..., GT_n\}$. The AMSG for MH is $\gamma - acyclic$ and $\lambda - acyclic$ so by Theorem 4.1 of Barker [6], MH is MDBSR.

Assume MH is not AMDBSR. This implies MH is equivalent to a history which satisfies conditions 1 and 2 of Definition 3.5.2 (because MH is MDBSR) but not conditions 3 or 4. Suppose condition 3 is violated. Then, the following are true: there exist GT_i, GT_j where GT_j is causally dependent on GT_i, GT_j is serialized before GT_i in the multidatabase history and, GT_i and GT_j conflict, either directly or indirectly. Since GT_i and GT_j conflict, there is a $\lambda, \gamma - path$ from GT_j to GT_i in AMSG(MH). However, AMSG(MH) also contains a $\pi - path$ from GT_i to GT_j. This implies that AMSG(MH) contains a causal discrepancy, which we know to be false. Hence, point 3 cannot be violated.

Suppose point 4 is violated. This means there exist a parent GT_i, a child GT_j, and some other GT_k a non-descendent of GT_i where GT_i is serialized before GT_k and GT_k is serialized before GT_j. This implies that GT_i and GT_k conflict and that GT_k and GT_j conflict. Since they conflict there is a $\lambda, \gamma - path$ from GT_i to GT_k and from GT_k to GT_j. Hence, there is a $\lambda, \gamma - path$ from GT_i to GT_j and GT_k lies on this path. This implies that AMSG(MH) contains a family order discrepancy which is a contradiction. Thus, item 4 is not violated either. Hence, our original assumption that MH is not AMDBSR was false, as required.

(only if): Given that the history is AMDB-serializable we must show that the AMSG produced must be $\gamma - acyclic$, $\lambda - acyclic$, causal discrepancy free, and

[2] C(MH) is the committed history of a MDB schedule which includes the committed transactions in each local history and the global history. C(MH) includes $C(LH^1), C(LH^2), ..., C(LH^n)$ and C(GH) which are those GTs in GH that are committed.[6]

family order discrepancy free.

Since MH is AMDBSR it is also MDBSR. Hence, by Theorem 4.1 in Barker [6], AMSG(MH) is $\gamma - acyclic$ and $\lambda - acyclic$. Now, we need to show that AMSG(MH) contains no discrepancies.

Assume AMSG(MH) contains a causal discrepancy. Then there exists GT_i and GT_j nodes in AMSG(MH) with a $\pi - path$ from GT_i to GT_j and a $\lambda, \gamma - path$ from GT_j to GT_i. This implies that GT_j is causally dependent on GT_i, and GT_j was serialized before GT_i. This means that MH is not AMDBSR, which violates our assumption. Hence, our assumption that AMSG(MH) contains a causal discrepancy is incorrect.

Assume AMSG(MH) contains a family order discrepancy. Then there exists a $\lambda, \gamma - path$ from GT_i to GT_j (a child of GT_i) and at least one node GT_k on this path is a non-descendent of GT_i. So, GT_i was serialized before a non-descendent GT_k which was serialized before GT_j, a child of GT_i. This implies MH is not AMDBSR, which it is. Hence, our assumption that AMSG(MH) contains a family order discrepancy is incorrect. Therefore, AMSG(MH) is $\gamma - acyclic$, $\lambda - acyclic$, and causal discrepancy free. ■

3.8 Summary of Assumptions

This model makes several assumptions either implicitly or explicitly. This chapter is concluded by collecting these assumptions.

1. *Local Autonomy:* Each of the local DBMSs are assumed to be totally autonomous. This means that no modifications to any DBMSs is permitted. This also means that the DBMSs cannot directly communicate with each

other. An individual DBMS is capable of executing a transaction submitted to it from start to finish and will ensure that in the event of a failure that the DBMS is able to fully recover without any user input.

2. *Subtransactions:* Compilation techniques exist to decompose a global transaction submitted to an AMDMS into global subtransactions. Further, a global transaction may submit at most one global subtransaction to any one DBMS.

3. *Reliability:* We view the issues of reliability and recovery as orthogonal so failures are not considered. Reliability and recovery techniques described in Barker [6] should be extended to ensure successful execution in this environment but these issues are left as future research.

4. *Trigger termination:* This model allows global trigger firings to be nested to any depth so trigger executions can fire other triggers. This can lead to infinite nestings of trigger executions. For example, a trigger GTR_i fires another trigger GTR_j, which in turn causes GTR_i to fire, which triggers GTR_j, and so on. This work assumes this situation will not occur. Aiken *et al.* [2] define mechanisms for detecting when a trigger set may lead to nonterminating executions and for detecting the triggers at fault. Other ideas may be found in Voort and Siebes [40].

Chapter 4

Global Trigger Manager

The global trigger manager is responsible for trigger event detection and for ensuring that global triggers are setup and submitted properly. However, detecting trigger events in this environment poses many problems. Existing models use *event detectors* [26, 14, 15, 24] to detect events as transactions are executing on the database. However, in a multidatabase environment adding event detectors to individual DBMSs would require major modifications resulting in a serious violation of local autonomy.

The GTRM deals with this problem by modifying submitted GSTs to perform their own event detection. If a global trigger event occurs as a result of the execution of an operation of some GST then the GST must signal to the GTRM that the event has occurred. It is then the GTRMs responsibility to execute the global trigger(s) fired as a result of the events occurrence.

Figure 4.1 illustrates the major components used by the GTRM in the event detection and trigger execution processes. These include the Global Trigger Dictionary (GTRD), Event Log (EL), and Dependency Graph (DG).

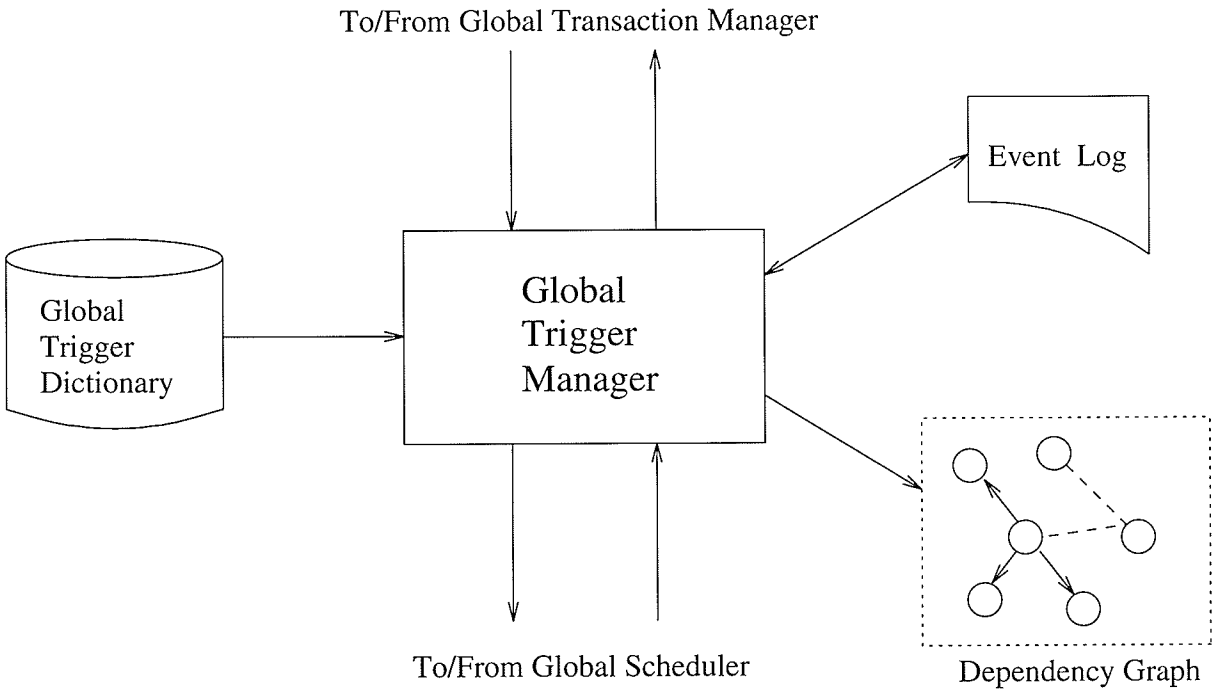

Figure 4.1: Illustration of the GTRM

Normally, the GTRM will setup a global transaction to execute the action of a trigger when the trigger event has been detected. The global transaction will then be submitted to the GS for scheduling. The GS will schedule the transaction as usual but will bear in mind any causal dependencies that may exist. The GRM will ensure that commit dependencies are ensured. However, there is one special case, the full dependency mode, when the GTRM will not be able to setup and submit global transactions. The next section addresses this situation.

4.1 Full Dependency Triggers

The full commit dependency coupling mode poses a problem in the multidatabase environment. Deadlock may arise if subtransactions of a triggering transaction and the triggered transaction execute concurrently at the same DBMS. We illustrate this problem with the following example.

Example 4.1.1 Our example uses a multidatabase consisting of three local databases. A global transaction GT_1 is submitted to the AMDMS and is parsed into two subtransactions, GST_1^1 and GST_1^2, which are submitted to the local databases. Suppose that GST_1^1 fires a global trigger GTR_1 which has a fully dependent coupling mode. The AMDMS creates $GT_{1.1}$ and its two subtransactions, $GST_{1.1}^2$ and $GST_{1.1}^3$, to execute the trigger action. Suppose that the global scheduler submits these GSTs to the local databases as well. $GST_{1.1}^2$ acquires a write lock on data item a at $DBMS^2$. GST_1^2 attempts to get a read lock on a but is suspended by the local scheduler and placed in the wait queue for data item a. $GST_{1.1}^2$ completes its operations and sends a *precommit* to the recovery manager. $GST_{1.1}^3$ also completes its operations and sends a precommit to the GRM. $GT_{1.1}$'s GSTs are all in a precommit state and is fully dependent so $GT_{1.1}$ will be committed if GT_1 commits.

However, GST_1^2 is waiting for $GST_{1.1}^2$ to give up a write lock, but it cannot until GT_1 commits. Hence, we have "self imposed" deadlock. ■

Deadlock can also arise even if the two GT's access independent sets of data. $GST_{1.1}^2$ may have a lock on a data item x and some local transaction LT_1^2 may enter the wait queue for the lock. Now, GST_1^2 enters the wait queue for a lock on y which LT_1^2 holds. $GST_{1.1}^2$ completes and precommits. LT_1^2 is still waiting for $GST_{1.1}^2$ to give up a lock and GST_1^2 is waiting on LT_1^2. However, $GT_{1.1}$ is waiting for GT_1 to commit. Again, we have deadlock.

One solution is to change the way the Global Scheduler and/or Global Recovery Managers work. We take the approach of modifying the GS and GRM as little as possible because proven algorithms exist for them that can extended to support active behavior [6].

An alternative solution is to *embed* the GSTs of full dependency GTs at the end of the GSTs of the triggering transaction. When the triggering transaction is complete the operations of the triggered GST may begin execution. If the operations of one GST dictate an abort then all of the GSTs are aborted. This is acceptable because the two transactions are commit dependent on each other. If the triggered transaction fires another full dependency trigger then this process recursively repeats itself. Hence, it is possible to get a GST that contains the operations of several subtransactions. This does, however, impose a serialization order for full dependency transactions. The embedding process is described in more detail when the algorithms are presented.

4.1.1 Embedding Issues

Embedding GSTs raises some interesting issues such as if a trigger is to execute zero times or more than once. The easier case is when a trigger is to fire zero times. If GST_i^j is embedded in some other GST_k^j and GT_i's event does not occur then the GST_i^j portion of GST_k^j must not execute. If GT_i fires two or more times then GST_i^j should execute the that number of times. These issues require that we embed the execution part of GSTs in a loop. At the beginning of each loop the GST will query the GTRM to determine the number of times to execute. The GTRM will know this because the triggering transaction had sent signals to it each time it causes a triggering event.

The next issue is when the loop of an embedded GST should terminate. The loop should complete when it is known that the triggering event of the embedded GST cannot occur again (i.e. the triggering transaction has completed). The GTRM knows that a triggering event will never occur again if enough of the triggering GTs subtransactions complete so that it is not possible for the event to by caused by the triggering GT. This requires that any subtransactions that cause simple events of GTRs must send a message to the GTRM notifying it that the triggering GST has completed its execution. At this point the GTRM examines each event that the GT may possibly cause and eliminates triggers that will never fire again. We call the message which notifies the GTRM of the GSTs completion a *termination*. Note that a GSTs abort or commit message is not sufficient for this purpose because we want the GTRM to know when a GST has completed its operations and if another GST is embedded in it then the GTRM would not be notified until the embedded GST completes.

Termination of GSTs is more complicated than it first seems. Consider the

following example.

Example 4.1.2 Suppose the execution of GST_1^1 could cause event e_1 to occur which triggers $GST_{1.1}^1$. Further, suppose that the execution of $GST_{1.1}^1$ could cause event e_2 to occur which fires $GST_{2.1}^1$. Both triggers are fully dependent so GTF_1 contains all three transactions. All three access the same DBMS so they are embedded into the same GST. Now suppose that GST_1^1 executes and does not cause e_1. It sends its termination to the GTRM which records it. $GST_{1.1}^1$ begins its loop and queries the GTRM for the number of times to execute. The GTRM knows that $GST_{1.1}^1$ is not to execute so replies with a message that the event occurred zero times and will never occur again. $GST_{1.1}^1$ is bypassed and $GST_{2.1}^1$ begins its loop. It queries the GTRM. The GTRM knows that e_2 has not occurred yet and also knows that the GT which causes it will never occur again. The GTRM informs $GST_{2.1}^1$ to execute zero times and complete. ■

The preceding example implies that the GTRM must check additional events when a GST terminates. That is, when GST_1^1 completes and sends its termination, the GTRM must mark the event which causes $GST_{1.1}^1$ (e_1), as final. Since e_1 never occurred the GTRM must mark all of the events caused by dependents of e_1 as final and set the number of occurrences to zero. $GST_{1.1}^1$ is execution dependent on e_1 and $GST_{1.1}^1$ causes e_2. Event e_2 must be marked as final. This is a recursive process until events are reached that have no dependents. When $GST_{2.1}^1$ queries in Example 4.1.2 the GTRM will know that it is to execute zero times.

Embedded GSTs which execute two or more times in a loop require extended data structures to record events caused by them. For example, suppose GST_1^1 could cause the event $conj(e_1, e_2)$ to occur. The first loop causes e_1 to occur and the second loop causes e_2. Semantically, $conj(e_1, e_2)$ has not occurred as the simple

events were caused by two logically separate GTs. Therefore, the GTRM must keep track of the events caused by different executions of a GST for trigger firing purposes.

4.2 Generic GST

The algorithms presented in this thesis assume that global subtransactions have a certain structure. Figure 4.2 illustrates the basic features of a GST submitted to a local database. The GST begins with a *database login* which logs onto the local database where the GST is submitted. Next, the *initialization* section sets up variables used by the transaction. The *transaction code* section performs the actual computations and database accesses. After the transaction code a precommit section sends a *ready* message to the AMDMS. The GST is then blocked until the AMDMS responds Once the response is received it commits or aborts according to the message. Finally, the *database logout* completes the GST. This approach is consistent with Barker [6].

4.3 GST Submission

Global transactions are parsed into their respected subtransactions by the global transaction manager. The set of GSTs are submitted as a group down the AMDS architecture to the GTRM (see Figure 1.2). This section describes the activities of the GTRM before the GSTs are passed to the global scheduler.

Global transactions are received and analyzed to determine if they could cause the event of one or more global triggers to occur. The GSTs of the submitted GT

```
Begin Transaction GST
    database login
    initialization
    transaction code
    send ready accept q_cond
    if q_cond == abort or send failed then
        abort
    else
        commit
    database logout
End Transaction GST
```

Figure 4.2: Generic Global Subtransaction

are then modified so they can send a message to the GTRM whenever they cause a simple event of a potential GTR. The GTRM will record event occurrences and control the execution of triggered GTs. If a potential GTR is fully dependent then a GT is created immediately for the GTR and added to the global transaction family of the submitted GT. This may involve embedding one or more GSTs (as discussed earlier). The intuitive initial GT submission algorithm follows.

Global Trigger Manager (*Initial GT_i Submission*)

1. Determine the global triggers that are potentially firable by GT_i.
2. Modify each GST of GT_i to signal the GTRM whenever the GST causes a simple event of a potential GTR.
3. For each potential GTR_k:
   (a) If GTR_k is not fully dependent then record GTR_k and its event in the event log.
   (b) If GTR_k is fully dependent then
      i. Modify each GST_i^j that causes one or more simple events of GTR_k to send a *termination* message when the GST has completed its operations.
      ii. Create a global transaction GT_h to execute GTR_k.
      iii. Record GT_h and its event in the event log.
      iv. For each $GST_h^r \epsilon GT_h$: if GT_h's family contains a GST_t^r then embed GST_h^r into GST_t^r; otherwise GST_h^r will be submitted with the GSTs of GT_i.
      v. Recursively call this algorithm with GT_h.

The following is a list of functions/data structures necessary for the description of the algorithms:

Dependency Graph : The dependency graph (DG) contains a node for each GT. A *commit-arc* from GT_i to GT_j is constructed if GT_i is commit dependent on GT_j. A *cause-arc* from GT_i to GT_j is constructed if GT_i is causally dependent on GT_j.

pending_GSTs (GT_i): This is a set of triggered GSTs that are potentially firable by GT_i. It is global in scope.

Add_Pending_GSTs (set_of_GSTs): Adds set_of_GSTs to the pending_GSTs list. This procedure modifies each GST so that it executes within a query loop (see the *Embed_GST* procedure described below).

event log : The event log (EL) is a stable log with six fields: *event*, *eventtree*, *occurrences*, *final*, *blocked*, and *dependents*. *Occurrences* records the number of times *event* has occurred, which is initially 0. *Eventtree* is an ordered set of event trees that record simple events (of *event*) which have occurred but have not yet caused *event* to occur. It initially contains only one eventtree but others may be added if this event is caused by a full dependency GT. *Final* is true if *event* will not occur again because transaction that causes one or more of the simple events has completed. The *blocked* field is a set of GSTs blocked on *event*. These GSTs may be started when *occurrences* is updated. *Dependents* is a set of global transaction or global trigger identifiers that are execution dependent on *event* so when *event* occurs each dependent is fired.

EL_Add (event_id): Adds a row to the event log. *Event* is set to event_id, *eventtree* is initialized to NULL, *occurred* is set to 0, *final* is initialized to false, *blocked*

is initialized to the empty queue, and *dependents* is initialized to the empty set. Only adds the row if event_id does not already occur in EL.

EL_Add_Dependent (id,event_id): Adds id to the *dependents* of the EL row with *event* = event_id.

TFT : This is transaction family tree that GT_i belongs to. It records the full dependencies between a GT and zero or more triggered GTs. It has a global scope.

TFT_Initialize (GT_i): Initializes the TFT to node GT_i.

TFT_DBMSs (): Returns a set of DBMS identifiers that are submitted to by GTs participating in the TFT.

TFT_GST (r): Performs an ordered traversal of the TFT searching for the first GT_j that submits a GST to $DBMS^r$. Returns the GST_j^r.

TFT_Add (GT_k,GT_i): Creates a node GT_k and adds it to the TFT as the right-most child of node GT_i.

Full_Dependency (GTR_k): Returns true if GTR_k has a full dependency coupling mode. Otherwise, returns false.

Generate_Potential_GTRs (set_of_GSTs): Generates and returns a set of GTRs which could directly fire as a result of the execution of set_of_GSTs.

GSTS (GT_h): Returns the set of GT_h's GSTs.

Event (GTR_k): Returns the event of GTR_k.

Insert_Transaction (GST_i^j,line,line_of_code): Inserts line_of_code after position line in the actual GST_i^j code.

Transaction_Code (GST_i^j): The transaction code of GST_i^j.

Initialization_Section (GST_i^j): The initialization section of GST_i^j.

Insert_Terminator (GST_i^j): Inserts a piece of code which we will call a terminator into GST_j^i to send a termination message to the GTRM when GST_i^j has completed its execution.

Simple_Events (GTR_k): Returns the set of simple events for the event of GTR_k.

Create_Global_Transaction (GTR_k): Creates a global transaction for the action of GTR_k. The dependency graph is updated with any commit and causal dependencies between the newly created transaction and its triggering transaction. Returns the new GT.

Figures 4.3, 4.4, 4.5, and 4.6 present the algorithms. Global transactions submitted to the AMDMS are parsed into their respective subtransactions by the global transaction manager and are passed down to the global trigger manager. Figure 4.3 presents the submission algorithm which accepts the GSTs and performs the necessary modifications for event detection and trigger execution. Line (1) sets the transaction family tree for GT_i to a graph containing only GT_i which is the global transaction of the GSTs submitted. Line (2) initializes pending_GSTs to the empty set. Line (3) calls a routine to modify the GSTs to detect simple events of global triggers which may fire as a result of GT_i's execution (see Figure 4.4). Finally, line (4) submits the GSTs of GT_i, and GSTs of full GTRs which may be fired by GT_i, to the global scheduler.

Figure 4.4 presents the recursive GST modification procedure. Line (1) generates the set of global triggers that could potential_GTRsly fire as a result of GT_i's execution. Line (2) creates the *monitor_set* which is the union of the sets of simple

Algorithm 4.1 *(Global Trigger Manager - Initial Submit)*

begin

input GT_i : global transaction identifier;
set_of_GSTs : **set of** GST;

TFT_Initialize(GT_i); (1)
pending_GSTs $\leftarrow \phi$; (2)
GST_Trigger_Modification(set_of_GSTs); (3)
Submit_GS(Set_of_GSTs $\bigcup$ pending_GSTs); (4)

end;

Figure 4.3: Global Trigger Manager Submission

events for each GTR that is potential_GTRsly firable. Lines (3-4) add each trigger's event to the event log, concatenated with GT_i's identifier. The concatenation will distinguish this event from the same event caused by some other global transaction. Lines (5-6) modify each GST of GT_i so that each will inform the GTRM whenever they cause an event of monitor_set to occur. Lines (7-19) examine each potential_GTRs GTR. If GTR_i does not have a full dependency then lines (8-9) add GTR_i's identifier to the dependents field of the triggering event in the event log. If GTR_i has a full dependency then lines (10-11) examine each GST of GT_i and ensure there is a terminator at the end of each GST that causes one or more simple events of GTR_k. Line (12) creates a new global transaction (GT_h) for the trigger. Line (13) adds GT_h to the dependents field of the event log row with event $= GT_i : Event(GTR_k)$. Each GST_h^r of GT_h is examined lines (14-17). Line (15) determines if there is a GT in the transaction family tree with a GST that is to be submitted to the same site as GST_h^r. If there is then line (16) embeds GST_h^r into the GST. If there is not, then line (17) adds GST_h^r to pending_GSTs. Line (18) adds GT_h to the TFT as a child of GT_i. Line (19) recursively calls this routine using the GSTs of GT_h

The algorithm presented in Figure 4.4 uses two important subroutines. Figure 4.5 modifies a GST to signal the GTRM whenever a monitored simple event occurs. Line (1) loops through each line in the database operations section of GST_i^j. Line (2) loops through each simple event caused by the current line. Line (3) checks if the simple event is to be monitored and, if so, a line of code is inserted in GST_i^j after the line which causes the event (lines (4-6)). The inserted line will signal the GTRM of the event's occurrence. If the triggered GST is of full dependency then the signal will include the execution number.

Figure 4.6 presents the subroutine to embed one GST into another. Line (1)

```
Procedure GST_Trigger_Modification;
begin
    input set_of_GSTs : set of GST;
    var potential_GTRs : set of GTR;
    var monitor_set : set of simple events;
    potential_GTRs ← Generate_Potential_GTRs(set_of_GSTs);                    (1)
    monitor_set ← ⋃k Simple_Events(GTRk), GTRk∈ potential_GTRs;               (2)
    for each GTRk∈ potential_GTRs do                                          (3)
        EL_Add(GTi:Event(GTRk));                                              (4)
    for each GSTij∈ Set_of_GSTs do                                            (5)
        Monitor_GST(GSTij,Monitor_set);                                       (6)
    for each GTRk∈ potential_GTRs do begin                                    (7)
        if not Full_Dependency(GTRk) then                                     (8)
            EL_Add_Dependent(GTRk,GTi:Event(GTRk));                           (9)
        else begin
            for each GSTij∈ set_of_GSTs do                                    (10)
                Terminator_GST(GSTij,Simple_events(GTRk));                    (11)
            GTh ← Create_Global_Transaction(GTRk);                            (12)
            EL_Add_Dependent(GTh,GTi:Event(GTRk));                            (13)
            for each GSThr∈GTh do                                             (14)
                if DBMSr ∈ TFT_DBMSs() then                                   (15)
                    Embed_GST(GSThr,TFT_GST(r));                              (16)
                else
                    Add_Pending_GSTs(GSThr);                                  (17)
                TFT_Add(GTh,GTi);                                             (18)
                GST_Trigger_Modification(GSTS(GTh));                          (19)
        end; (* else *)
    end; (* for *)
end;
```

```
Procedure Monitor_GST(GST_i^j, set_of_events);

begin

    var
        line : line of transaction code;

    for each line in Transaction_Code(GST_i^j) do                          (1)
        for each simple event e caused by line do                          (2)
            if e∈ set_of_events then                                       (3)
                if not Full(GST_i^j) then                                  (4)
                    Insert_Transaction(GST_i^j,line,"Signal_GTRM(GST_i^j : e,1);");   (5)
                else
                    Insert_Transaction(GST_i^j,line,"Signal_GTRM(GST_i^j : e,loop);");  (6)
end;
```

Figure 4.5: GST monitor procedure

finds the line used to precommit in the GST which will have another GST embedded. Line (2) embeds the relevant GST code before the original GST's precommit.

The next example provides a detailed trace of the submission algorithm's execution.

Example 4.3.1 Our example AMDS consists of five local databases: $\mathcal{LDB}^1 = \{d, e\}$, $\mathcal{LDB}^2 = \{x, y\}$, $\mathcal{LDB}^3 = \{n, m\}$, $\mathcal{LDB}^4 = \{s, t\}$, and $\mathcal{LDB}^5 = \{p, q\}$. The global trigger dictionary contains three GTRs:

$$GTR_1 : (\{r(d); w(d); w(x); w(y); w(m); c; \}, r(e), Fdep)$$
$$GTR_2 : (\{w(d); r(p); w(q); c; \}, w(s), Cdep)$$
$$GTR_3 : (\{r(y); w(x); w(n); w(p); c; \}, conj(w(x), w(d)), Fdep)$$

A single global transaction is submitted to the AMDMS:

$$GT_1 : r_1(e); w_1(s); w_1(y); w_1(d); c_1;$$

GT_1's GSTs are received by Algorithm 4.1:

$$GST_1^1 : r_1^1(e); w_1^1(d); c_1^1;$$
$$GST_1^2 : w_1^2(y); c_1^2;$$
$$GST_1^4 : w_1^4(s); w_1^4(t); c_1^4;$$

The GTRM's algorithms are illustrated by the following execution sequence:

1. The TFT is initialized to node GT_1 and pending_GSTs is initialized to the empty set (lines (1-2)). The routine to perform trigger modifications is called for the GSTs of GT_1 (line (3)).

Procedure Embed_GST(GST_k^r, GST_i^j, event);

begin

var

line : line of transaction code;

line ← Precommit_line(GST_i^j); (1)

Insert_Transaction(GST_i^j,line-1, (2)

"{

"+Initialization_Section(GST_k^r)+

" int count_loop, loop, final = 0;

do

count_loop, final = Query_GTRM(GST_k^r,loop)

while loop < count_loop do

{

"+Transaction_Code(GST_k^r)

+"loop++

}

while !final

}

");

end;

Figure 4.6: GST embedding routine

```
Begin Transaction GST
    database login
    initialization
    transaction code
    {
        // begin embedded GST_i^j
        initialization
        int count_loop, loop, final = 0
        do
            count_loop, final = Query_GTRM(GST_i^j,loop)
            while loop < count_loop do
            {
                transaction code
                loop++
            }
        while !final
        // end embedded GST_i^j
    }
    send ready accept q_cond
    if q_cond == abort or send failed then
        abort
    else
        commit
    database logout
End Transaction GST
```

Figure 4.7: Embedded GST

```
Procedure Terminator_GST(GST_i^j, set_of_events);

begin
    var
        line : line of transaction code;

    if GST_i^j does not have a terminator then
        for each line in Transaction_Code(GST_i^j) do
            if line causes an event of set_of_events then
            begin
                Insert_Terminator(GST_i^j);
                return;
            end;
end;
```

Figure 4.8: Terminator insertion routine

The potential_GTRs set of triggers firable by GT_1 is generated in the trigger modifications procedure. The GTR events include $r(e)$, $w(s)$, and $conj(w(x), w(d))$. The simple events are $r(e)$, $w(s)$, $w(x)$, and $w(d)$. GT_1 may cause all but $w(x)$ to occur. Hence, the event $conj(w(x), w(d))$ will not occur as a result of $GT_1's$ execution and GTR_3 will not fire. However, GTR_1 and GTR_2 may fire. Hence, we get:

$$\text{potential\_GTRs} = \{GTR_1, GTR_2\}$$

The set of all simple events of the potential GTRs is generated (line (2)):

$$\text{monitor\_set} = \{r(e), w(s)\}$$

Each trigger's event is added to the event log (lines (3-4)) with the GT_1 identifier (see line (1) of Figure 4.9 which depicts the event log).

2. Each GST of GT_1 is modified to detect the simple events of monitor_set (lines (5-6)). This involves a call to Monitor_GST. The line:

   Signal_GTRM($GST_1^1 : r(e)$,1);

   is inserted into GST_1^1 after the $r(e)$ operation and the line:

   Signal_GTRM($GST_1^4 : w(s)$,1);

   is inserted into GST_1^4 after the $w(s)$ operation. When these lines are executed by the GST, a message is sent to the GTRM with the event that has occurred. The GTRM can then execute GTR_1 and GTR_2, respectively. Each trigger in potential_GTRs is examined, starting with GTR_1 (line (7)). GTR_1 is a full

dependency trigger (line (8)) so a terminator is added to GST_1^1 (lines (10-11)). A new global transaction $GT_{1.1}$ for GTR_1 is created (line (12)) whose GSTs are:

$$GST_{1.1}^1 : r_{1.1}^1(d); w_{1.1}^1(d); c_{1.1}^1;$$
$$GST_{1.1}^2 : w_{1.1}^2(x); w_{1.1}^2(y); c_{1.1}^2;$$
$$GST_{1.1}^3 : w_{1.1}^3(m); c_{1.1}^3;$$

Information is added to the event log that indicates $GT_{1.1}$ is execution dependent on the event $GT_1 : r(e)$ (line (13)). Line (2) of Figure 4.9 depicts the event log at this point.

3. $GST_{1.1}^1$ is checked to see if the transaction family contains a GST to be submitted to $DBMS^1$ ($DBMS^2$) (line (15)). The TFT contains only node GT_1 which visits $DBMS^1$, $DBMS^2$, and $DBMS^4$ so TFT_DBMSs = $\{DBMS^1$, $DBMS^2$,$DBMS^4\}$. $GST_{1.1}^1$ ($GST_{1.1}^2$) is embedded into GST_1^1 (GST_1^2) (line (16)). $GST_{1.1}^3$ accesses $DBMS^3$ which is not currently accessed by GTs of the TFT so $GST_{1.1}^3$ is added to pending_GSTs (line (17)). Node $GT_{1.1}$ is added to the TFT as the rightmost child of GT_1 (line (18)). The TFT now contains the node GT_1 and its child $GT_{1.1}$. The GST trigger modification routine is called (recursively) with the GSTs of $GT_{1.1}$ to determine the triggers which may fire due to its execution (line (19)).

   The recursive call begins by generating the set of potential triggers for $GT_{1.1}$. The only trigger event that is possible is the $conj(w(x), w(d))$ event of GTR_3. Hence:

$$\text{potential\_GTRs} = \{GTR_3\}$$

   The set of all simple events of GTR_3 is generated:

$$\text{monitor\_set} = \{w(x), w(d)\}$$

The event log is updated and line (3) of Figure 4.9 depicts the situation.

4. Each GST of $GT_{1.1}$ are now modified to detect the simple events of monitor_set. GTR_3 is a full dependency trigger so a terminator is added to $GST^1_{1.1}$ and $GST^2_{1.1}$. A new global transaction $GT_{3.1}$ is created for GTR_3 and the GSTs for $GT_{3.1}$ are:

$$GST^2_{3.1} : r^2_{3.1}(y); w^2_{3.1}(x); c^2_{3.1};$$
$$GST^3_{3.1} : w^3_{3.1}(n); c^3_{3.1};$$
$$GST^5_{3.1} : w^5_{3.1}(p); c^5_{3.1};$$

Information is added to the event log that indicates $GT_{3.1}$ is execution dependent on the event $GT_{1.1} : conj(w(x), w(d))$. The updated event log appears on line (4) of Figure 4.9.

5. Next, each GST for $GT_{3.1}$ is examined to determine whether the TFT currently contains a GT which is sending to that site. The TFT now contains nodes GT_1 and $GT_{1.1}$ and so TFT_DBMSs = $\{DBMS^1, DBMS^2, DBMS^3, DBMS^4\}$. $GST^2_{3.1}$ is embedded into GST^2_1 and $GST^3_{3.1}$ is embedded in $GST^3_{1.1}$. $DBMS^5$ is not part of TFT_DBMSs so $GST^5_{3.1}$ is added to pending_GSTs.

The node $GT_{3.1}$ is added to the TFT as a right sibling of $GT_{1.1}$'s children. The TFT now contains the node GT_1, its child $GT_{1.1}$, and $GT_{1.1}$'s child $GT_{3.1}$. The GST modification routine is recursively called with the GSTs of $GT_{3.1}$ but returns quickly as there are no triggers potentially firable by $GT_{3.1}$.

This GST_Trigger_Modification call returns to the caller (the GST trigger modification executing on GT_1). The second potential trigger for GT_1 (GTR_2)

	event	eventree	occurred	final	blocked	dependents
(1)	$GT_1 : r(e)$	ϕ	0	$false$	ϕ	ϕ
	$GT_1 : w(s)$	ϕ	0	$false$	ϕ	ϕ
(2)	$GT_1 : r(e)$	ϕ	0	$false$	ϕ	$\{GT_{1.1}\}$
	$GT_1 : w(s)$	ϕ	0	$false$	ϕ	ϕ
(3)	$GT_1 : r(e)$	ϕ	0	$false$	ϕ	$\{GT_{1.1}\}$
	$GT_1 : w(s)$	ϕ	0	$false$	ϕ	ϕ
	$GT_{1.1} : conj(w(x), w(d))$	ϕ	0	$false$	ϕ	ϕ
(4)	$GT_1 : r(e)$	ϕ	0	$false$	ϕ	$\{GT_{1.1}\}$
	$GT_1 : w(s)$	ϕ	0	$false$	ϕ	ϕ
	$GT_{1.1} : conj(w(x), w(d))$	ϕ	0	$false$	ϕ	$\{GT_{3.1}\}$
(5)	$GT_1 : r(e)$	ϕ	0	$false$	ϕ	$\{GT_{1.1}\}$
	$GT_1 : w(s)$	ϕ	0	$false$	ϕ	$\{GTR_2\}$
	$GT_{1.1} : conj(w(x), w(d))$	ϕ	0	$false$	ϕ	$\{GT_{3.1}\}$

Figure 4.9: Event Log Trace for Example 4.3.1

does not have a full dependency (line (8)) and GTR_2 is added to the dependents field of the row in the event log with event = $GT_1 : w(s)$. Line (5) of Figure 4.9 illustrates the event log. The GST trigger modification subroutine completes and returns to Algorithm 4.1. Algorithm 4.1 finishes by submitting GST_1^1, GST_1^2, and GST_1^4 as well as the pending $GST_{1.1}^3$ and $GST_{3.1}^5$ to the global scheduler.

■

4.4 Signal, Query, and Termination Handling

The process of trigger management does not stop at GST submission to the global scheduler. Communication between the GTRM and executing GSTs is an important part of trigger management in this environment. This section describes the signal, query, and termination handling techniques.

While subtransactions are executing on DBMSs they must *signal* the GTRM whenever simple events are triggered. Embedded GSTs must *query* the GTRM for the number of times to execute. Finally, GSTs that cause simple events of full dependency GTRs must send their termination so the GTRM can inform waiting GSTs that an event will not occur again. Each of these concepts is described intuitively below:

Global Trigger Manager (*Signal Handling*)

1. Send acknowledgement to sending GST so that it may resume its execution.
2. For each event in the event log that may occur due to the simple event signaled:
   (a) Update the event log entry.
   (b) If event of the EL occurs then
      i. Increment the number of occurrences of this event.
      ii. Send the number of occurrences of the event to all GSTs that are currently blocked on it. Remove these GSTs from the blocked queue for this event.

iii. Create a global transaction GT_h for each GTR that is execution dependent on this events occurrence. Call the initial submission algorithm with the GSTs of GT_h.

Global Trigger Manager (*Query Handling*)

1. Find the event in the event log that the querying GST is dependent on.
2. If the event will not occur again then send the number of occurrences and a *final* message to the querying GST. Terminate.
3. Otherwise, if the querying GST has executed the same number of times as the number of times the event has occurred then add the sender to the blocked queue of the event.
4. If the actual number of event occurrences is greater than the number of times the querying GST has already executed then send the actual number of event occurrences to the GST.

Before presenting the intuitive termination handler we define what is meant by a *necessary simple event* of a complex event.

Definition 4.4.1 *(Necessary simple event):* Let e_i be a simple event of some complex event e_j. We say e_i is *necessary* if e_i must occur for e_j to occur. ■

For example, the complex event conj(e_1, e_2) is composed of the simple events e_1 and e_2 both of which are necessary for conj(e_1, e_2) to occur.

In the algorithm below, if GST_j^i terminates then all events that are caused by GT_j are checked for necessary simple events caused by GST_j^i. If one or more are found then the event will never occur again and can be marked as final.

Global Trigger Manager (*Termination Handling*)

1. For each event in the event log which contains *necessary simple events* caused by the terminating GST:

   (a) Mark the event as being final.

   (b) Send each GST that is blocked on the current event the number of occurrences of the event and a final message. Remove the GST from the blocked queue.

   (c) If the current event never occurred then for each identifier i that is execution dependent on this event, mark the events that i causes (in the event log) as final and perform step (b) on any blocked GSTs. Recursively perform step (c) on the events caused by dependents of the current event.

The algorithms that handle event signals, queries, and terminations are presented in Figures 4.10, 4.11, and 4.12, respectively. The following is a list of additional functions/data structures necessary for the description of the algorithms:

signal (GST,event,execution number): Message sent from a GST to the GTRM when some event occurs. A signal contains the event that occurred, the execution number, and the GST that caused it.

query (GST,times_to_date): Message sent from a GST to the GTRM when the GST needs to know how many times to execute. A *query* contains the *GST* that is querying and the number of times that it has executed so far.

termination : Message sent from a GST to the GTRM when the sending GST has completed its execution.

Event (msg): Returns the event of the *msg* signal.

Sender (msg): Returns the GST which sent *msg*.

Executions (msg): Returns the times_to_date of the query *msg*.

GT (msg): Returns the GT of the sender of *msg*.

When a GST signals the GTRM of an event's occurrence, Algorithm 4.2 of Figure 4.10 is invoked. First an acknowledgement is sent to the signaling GST that notifies the GST to resume its execution (line (1)). Each row in the event log is considered in turn (line (2)). If the event of the signal is a simple event of the current row's event (line (3)) then the eventtree of the current row is updated. The execution number of the message is used to determine the correct eventtree to update (line (4)). If the eventtree for the execution number does not exist then one is created and updated. If the updated eventtree is occurs then the eventtree is reset (line (6)) the number of event occurrences is incremented (line (7)). Each GST_i^j that was blocked on the event is sent the new number of occurrences and the final flag and is removed from the blocked queue (lines (8-10)).

Line (11) loops through each global trigger identifier in the dependents field of the row. Line (12) creates a new global transaction for the GTR and line (13) submits it to Algorithm 4.1 to receive trigger modifications. Algorithm 4.1 is executed by a separate, concurrent process.

The GTRM query handler is much simpler. When a GST queries the GTRM for the number of times to execute Algorithm 4.3 is invoked (illustrated in Figure 4.11). The row in the EL which the sending GT is dependent on is found (line (1)). If the number of occurrences of the event is equal to the number of executions of the querying GST (line (2)) then the GST has executed the required number of times

Algorithm 4.2 *(Global Trigger Manager - Signal Handler)*

```
begin
    input msg : signal;
    send ack to Sender(msg);                                              (1)
    for each row r in EL do                                               (2)
    begin
        if Event(msg) ϵ row.event and Sender(msg) = GT(row.event) then    (3)
        begin
            update r.eventtree with Event(msg) and Executions(msg);       (4)
            if r.eventtree occurs then                                    (5)
            begin
                reset the r.eventtree;                                    (6)
                r.occurred ← r.occurred + 1;                              (7)
                for each GST_i^j ϵ r.blocked do                           (8)
                begin
                    send r.occurred and r.final to GST_i^j;               (9)
                    remove GST_i^j from r.blocked;                        (10)
                end;
                for each i ϵr.dependents where i is a GTR identifier do   (11)
                begin
                    GT_h ← Create_Global_Transaction(i);                  (12)
                    pipe to Algorithm 4.1 with h and GSTS(GT_h);          (13)
                end; (* for *)
            end; (* if *)
        end; (* if *)
    end; (* for *)
end;
```

Figure 4.10: Global Trigger Manager Signal Handler

Algorithm 4.3 *(Global Trigger Manager - Query Handler)*

```
begin
    input msg : query;
    find row r in EL with GT(msg) ϵ r.dependents;                    (1)
    if r.occurred = Executions(msg) and not r.final then             (2)
        add Sender(msg) to r.blocked;                                (3)
    else
        send r.occurred and r.final to Sender(msg);                  (4)
end;
```

Figure 4.11: Global Trigger Manager Query Handler

and is added to the blocked queue of the event in the EL (line (3)). If not, the number of occurrences and the final flag of the event are sent to the querying GST (line (4)) which executes the desired number of times.

The GTRM termination handler is also quite simple. When a GST sends its termination to the GTRM, Algorithm 4.4 of Figure 4.12 is called. Each event in the EL which is caused by the GT that sent the termination is considered in turn (line (1)). If the event of the EL contains necessary simple events caused by the terminated GST (line (2)) then the final flag of the event is set to true (line (3)). Each GST that is blocked on the event (line (4)) is sent the number of occurrences and final flag (line (5)) and is removed from the blocked queue (line (6)). If the event has never occurred then procedure Terminate is called with r.dependents (lines (7-8)).

Terminate (Figure 4.13) is a procedure that recursively updates events caused by terminated GTs that never executed. Each GT identifier in set_of_IDs is considered. Each event in the event log which is caused by the current GT is checked (line (2)). The final flag is set to true and the blocked GSTs are updated (lines (3-6)) as in Algorithm 4.4. Terminate is recursively called with the dependents of the current row (line (7)). This way, all events that will never occur (as a result of some GSTs termination) will be properly marked as final with zero occurrences.

The next example provides a detailed trace of signal, query, and termination algorithm executions.

Example 4.4.1 Recall 4.3.1. Suppose that the subtransactions of GTF_1 (GT_1, $GT_{1.1}$, and $GT_{3.1}$) have been successfully scheduled by the global scheduler and are executing at the corresponding sites. We use the execution of these GSTs to demonstrate the functionings of Algorithms 4.2, 4.3, and 4.4. We will limit our example to the GSTs presented in Example 4.3.1. Of course, there could be local transactions and GSTs of other active GTFs executing at the local databases, but we keep the example simple. Figure 4.14 illustrates the subtransactions at each local database.

1. GST_1^2 is the first to complete. $GST_{1.1}^2$, which is embedded in GST_1^2, makes a query to the GTRM, indicating that it has executed zero times so far. Algorithm 4.3 is invoked by the GTRM and finds the entry for $GT_{1.1}$ in the event log, checking if the number of occurrences in the event log matches the number of executions of $GST_{1.1}^2$. The number of occurrences is zero which is equal to the number of executions, and the event is not final so $GST_{1.1}^2$ is added to the blocked queue of the event (line (3)). The GTRM does not have to actually block $GST_{1.1}^2$ as it has suspended itself as a result of this query.

Algorithm 4.4 *(Global Trigger Manager - Termination Handler)*

```
begin
    input msg : termination;
    for each row r in EL with GT(r.Event) = GT(msg) do                    (1)
        if r.event contains necessary simple events caused by Sender(msg) then  (2)
        begin
            r.final ← true;                                               (3)
            for each GST_i^j ϵ r.blocked do                               (4)
            begin
                send r.occurred and r.final to GST_i^j;                   (5)
                remove GST_i^j from r.blocked;                            (6)
            end;
            if r.occurred = 0 then                                        (7)
                Terminate(r.dependents);                                  (8)
        end;
end;
```

Figure 4.12: Global Trigger Manager Termination Handler

```
Procedure Terminate(set_of_IDs);

begin
    for each GT identifier i ϵ set_of_IDs do                    (1)
        for each row r in EL with GT(r.Event) = i do            (2)
        begin
            r.final ← true;                                     (3)
            for each GST_i^j ϵ r.blocked do                     (4)
            begin
                send r.occurred and r.final to GST_i^j;         (5)
                remove GST_i^j from r.blocked;                  (6)
            end;
            Terminate(r.dependents);                            (7)
        end;
end;
```

Figure 4.13: Procedure Terminate

	GTF_1	$GTF_{2.1}$
$DBMS^1$:	$(GST^1_{1(1)}, GST^1_{1.1(1)})$	$GST^1_{2.1}$
$DBMS^2$:	$(GST^2_{1(1)}, GST^2_{1.1(1)}, GST^2_{3.1(1)})$	
$DBMS^3$:	$(GST^3_{1.1(1)}, GST^3_{3.1(1)})$	
$DBMS^4$:	$GST^4_{1(1)}$	
$DBMS^5$:	$GST^5_{3.1(1)}$	$GST^5_{2.1}$

Figure 4.14: Subtransactions Executing At Each Site

This completes Algorithm 4.3. Line (1) of Figure 4.15 depicts the event log at this point.

2. $GST^5_{3.1}$ is next to query to the GTRM, indicating that it has executed zero times so far. Algorithm 4.3 finds the entry for $GT_{3.1}$ in the event log. The number of occurrences is zero as is the number of executions, and the event is not final so $GST^5_{3.1}$ is added to the blocked queue of the event. Line (2) of Figure 4.15 illustrates the event log.

3. Next, GST^1_1 executes its $r(e)$ operation and signals the GTRM of its occurrence which invokes Algorithm 4.2. GST^1_1 is sent an acknowledgement to resume its execution. The event that was signaled is checked to see if it is a simple event of the current event of the event log. It is for the event $GT_1 : r(e)$. The eventtree of $GT_1 : r(e)$ is updated and $GT_1 : r(e)$ is checked for occurrence. The event is the simple event $r(e)$ caused by GT_1 so it has occurred. The eventtree is reset and the number of occurrences is incremented. Each GST that is blocked on the event is sent the number of occurrences and is removed from the queue (lines (8-10)). $GST^2_{1.1}$ is the only one and so begins

its first execution. Line (3) of Figure 4.15 depicts the event log.

4. Next, GST_1^4 signals that it has executed its $w(s)$ operation. Algorithm 4.2 sends an acknowledgement to GST_1^4 and checks to see if the signaled event is a simple event for one or more rows in the event log It is for $GT_1 : w(s)$. The eventtree of $GT_1 : w(s)$ is updated and has occurred. The eventtree is reset and the number of occurrences is incremented. GTR_2 is the only dependent global trigger identifier (line (11)) so a new global transaction $GT_{2.1}$ is setup for the newly fired trigger. The GSTs of $GT_{2.1}$ are sent to Algorithm 4.1 for trigger modification. No modifications are necessary and $GT_{2.1}$'s GSTs are sent to the global scheduler. To keep this example simple suppose the GS submits $GST_{2.1}^1$ and $GST_{2.1}^5$ immediately. Line (4) of Figure 4.15 depicts the event log.

5. GST_1^1 executes its $w(d)$ operation next and sends its termination to the GTRM which invokes Algorithm 4.4. Each event in the event log that is caused by GT_1 is considered and $GT_1 : r(e)$ is the first. $GT_1 : r(e)$ is checked for necessary simple events caused by GST_1^1 of which $r(e)$ is the only one (line (2)). This event can never occur again so the final flag of $GT_1 : r(e)$ is set to true. There are no GSTs blocked on the event so $GT_1 : w(s)$ is considered next but does not contain simple events caused by GST_1^1. This completes the algorithm. Line (5) of Figure 4.15 depicts the situation.

6. Next, $GST_{1.1}^1$ queries invoking Algorithm 4.3. The entry in the event log for $GT_{1.1}$ is found . The number of executions of $GST_{1.1}^1$ is one and the number of executions of $GST_{1.1}^1$ is zero so the number of occurrences and the final flag are sent to $GST_{1.1}^1$ and Algorithm 4.3 completes. GST_1^4 executes its $w(t)$ operation and precommits. $GST_{1.1}^2$ executes its $w(x)$ operation and signals

this event to the GTRM invoking Algorithm 4.2 which determines that $w(x)$ is a simple event of $GT_{1.1} : conj(w(x), w(d))$. The eventtree is updated but $GT_{1.1} : conj(w(x), w(d))$ has not occurred (line (5)). In this case, the eventtree still lacks the $w(d)$ event, so it is not satisfied and Algorithm 4.2 terminates. Line (6) of Figure 4.15 depicts the event log.

7. $GST^1_{1.1}$ executes its $r(d)$ and $w(d)$ operations next, and signals the GTRM of the $w(d)$ event which invokes Algorithm 4.2. The eventtree of $GT_1 : conj(w(x), w(d))$ is updated and both $w(d)$ and $w(s)$ have occurred so the eventtree is satisfied. The eventtree is reset and the number of occurrences is incremented. $GST^5_{3.1}$ is the only blocked GST so it is sent the new occurrences and is removed from the queue. $GST^5_{3.1}$ begins its first execution. Line (7) of Figure 4.15 illustrates the situation.

8. Next, $GST^3_{1.1}$ executes its $w(m)$ operation and queries the GTRM invoking Algorithm 4.3. The entry for $GT_{1.1}$ in the event log is found and the event is final. The number of occurrences and the final flag are sent to $GST^1_{1.1}$. Algorithm 4.3 completes and $GST^3_{1.1}$ completes. Meanwhile, $GST^1_{2.1}$ and $GST^5_{2.1}$ execute and commit independent of GT_1, $GT_{1.1}$, and $GT_{3.1}$. $GST^3_{3.1}$ queries, is sent the number of occurrences (1), the final flag (false) and begins its first execution. $GST^1_{1.1}$ finishes its first execution and knows that it is supposed to execute only once. It is complete and sends its termination to the GTRM invoking Algorithm 4.4. $GT_{1.1} : conj(w(x), w(d))$ is the only event caused by $GT_{1.1}$ and contains necessary simple events caused by $GST^1_{1.1}$, namely $w(d)$. The final flag of $GT_{1.1} : conj(w(x), w(d))$ is set to true. There are no blocked GSTs and the event did occur (lines (4-8)) so the Algorithm terminates. Line (8) of Figure 4.15 depicts the event log.

9. Next, $GST^2_{1.1}$ finishes its first execution and queries. The GTRM responds with one execution and a true final flag. $GST^2_{1.1}$ is complete and sends its termination, which was necessary for the monitoring of $GT_{1.1} : conj(w(x), w(d))$. $GST^2_{3.1}$ queries and is replied to with one execution and a true final flag. It begins its only execution. Next, $GST^5_{3.1}$ and $GST^3_{3.1}$ complete their execution and query receiving one execution and a true final flag. They are complete. $GST^2_{3.1}$ executes its operations and precommits. At this point all GSTs have precommitted and the recovery manager handles global transaction commitment.

■

	event	eventree	occurred	final	blocked	dependents
(1)	$GT_1 : r(e)$	ϕ	0	$false$	$\{GST^2_{1.1}\}$	$\{GT_{1.1}\}$
	$GT_1 : w(s)$	ϕ	0	$false$	ϕ	$\{GTR_2\}$
	$GT_{1.1} : conj(w(x), w(d))$	ϕ	0	$false$	ϕ	$\{GT_{3.1}\}$
(2)	$GT_1 : r(e)$	ϕ	0	$false$	$\{GST^2_{1.1}\}$	$\{GT_{1.1}\}$
	$GT_1 : w(s)$	ϕ	0	$false$	ϕ	$\{GTR_2\}$
	$GT_{1.1} : conj(w(x), w(d))$	ϕ	0	$false$	$\{GST^5_{3.1}\}$	$\{GT_{3.1}\}$
(3)	$GT_1 : r(e)$	ϕ	1	$false$	ϕ	$\{GT_{1.1}\}$
	$GT_1 : w(s)$	ϕ	0	$false$	ϕ	$\{GTR_2\}$
	$GT_{1.1} : conj(w(x), w(d))$	ϕ	0	$false$	$\{GST^5_{3.1}\}$	$\{GT_{3.1}\}$
(4)	$GT_1 : r(e)$	ϕ	1	$false$	ϕ	$\{GT_{1.1}\}$
	$GT_1 : w(s)$	ϕ	1	$false$	ϕ	$\{GTR_2\}$
	$GT_{1.1} : conj(w(x), w(d))$	ϕ	0	$false$	$\{GST^5_{3.1}\}$	$\{GT_{3.1}\}$
(5)	$GT_1 : r(e)$	ϕ	1	$true$	ϕ	$\{GT_{1.1}\}$
	$GT_1 : w(s)$	ϕ	1	$false$	ϕ	$\{GTR_2\}$
	$GT_{1.1} : conj(w(x), w(d))$	ϕ	0	$false$	$\{GST^5_{3.1}\}$	$\{GT_{3.1}\}$
(6)	$GT_1 : r(e)$	ϕ	1	$true$	ϕ	$\{GT_{1.1}\}$
	$GT_1 : w(s)$	ϕ	1	$false$	ϕ	$\{GTR_2\}$
	$GT_{1.1} : conj(w(x), w(d))$	$w(x)$	0	$false$	$\{GST^5_{3.1}\}$	$\{GT_{3.1}\}$
(7)	$GT_1 : r(e)$	ϕ	1	$true$	ϕ	$\{GT_{1.1}\}$
	$GT_1 : w(s)$	ϕ	1	$false$	ϕ	$\{GTR_2\}$
	$GT_{1.1} : conj(w(x), w(d))$	ϕ	1	$false$	ϕ	$\{GT_{3.1}\}$
(8)	$GT_1 : r(e)$	ϕ	1	$true$	ϕ	$\{GT_{1.1}\}$
	$GT_1 : w(s)$	ϕ	1	$false$	ϕ	$\{GTR_2\}$
	$GT_{1.1} : conj(w(x), w(d))$	ϕ	1	$true$	ϕ	$\{GT_{3.1}\}$

Figure 4.15: Event Log Trace for Example 4.4.1

Chapter 5

Global Scheduler

The architecture of Chapter 1 and formal model of Chapter 3 allow for modular design of concurrency control algorithms which ensure AMSB-serializability. This chapter presents one such algorithm. We modify Barker's AGSS (Aggressive Global Serial Scheduler) which ensures MDB-serializability so that the resulting algorithm ensures the more restrictive AMDB-serializability.

The remainder of this chapter proceeds as follows. Section 5.1 describes requirements of the scheduler to ensure AMDB-serializability. Section 5.2 presents an intuitive description of the scheduler and a more detailed pseudocode representation of the algorithm. An example and step by step walkthrough are given to demonstrate the execution of the algorithm. Finally, the algorithm is proven correct.

5.1 Scheduler Requirements

Barker [6] presents two concurrency control algorithms for the global scheduler each of which ensures MDB-Serializability. These algorithms control the order of GST submission in a pessimistic fashion. However, these do not ensure AMDB-Serializability. To see this we present two examples.

Example 5.1.1 Consider the following sequence of global transactions and their corresponding GSTs:

$$\begin{array}{lllll} GT_1 & : GST_1^1 & & & \\ GT_{1.1} & : & GST_{1.1}^2 & & \\ GT_2 & : & GST_2^2 & GST_2^3 & \\ GT_3 & : GST_3^1 & & GST_3^3 & \end{array}$$

Suppose that $GT_{1.1}$ is a full global transaction fired by GT_1. The following sequence is possible with the AGSS:

$$\begin{aligned} DBMS^1 &: GT_1 \prec GT_3 \\ DBMS^2 &: GT_2 \prec GT_{1.1} \\ DBMS^3 &: GT_3 \prec GT_2 \end{aligned}$$

which would produce an AMSG with a family order discrepancy where $GT_1 \rightarrow GT_3 \rightarrow GT_2 \rightarrow GT_{1.1}$. ■

The next example demonstrates that causal discrepancies are possible with the AGSS.

Example 5.1.2 Consider the following sequence of global transactions and their corresponding GSTs:

$$\begin{aligned} GT_1 &: GST_1^1 \\ GT_2 &: GST_2^1\, GST_2^2 \\ GT_{1.1} &: \qquad GST_{1.1}^2 \end{aligned}$$

Suppose that $GT_{1.1}$ has a causally dependent coupling mode and is fired by GT_1. The following sequence is possible with the AGSS:

$$\begin{aligned} DBMS^1 &: GT_2 \prec GT_1 \\ DBMS^2 &: GT_{1.1} \prec GT_2 \end{aligned}$$

which would produce an AMSG with the causal discrepancy $GT_{1.1} \rightarrow GT_2 \rightarrow GT_1$.

■

5.2 Active Global Scheduler for AMDB-Serializability

We extend Barker's AGSS to cope with the situations of Examples 5.1.1 and 5.1.2. This section describes the extended algorithm, which we refer to as the *Active Global Scheduler*(AGS). Intuition, pseudocode, examples, and correctness are presented in this section. Figure 5.1 illustrates the global scheduler. The GS receives GSTs from the GTRM. When the GS realizes that it is safe to submit them it passes the GSTs to the global recovery manager which submits the GSTs to the appropriate DBMSs. The GS uses the dependency graph created by the global trigger manager.

The AGS must ensure that the AMSGs of histories it produces do not contain any causal or family order discrepancies. We investigate Examples 5.1.1 and 5.1.2 for new ideas.

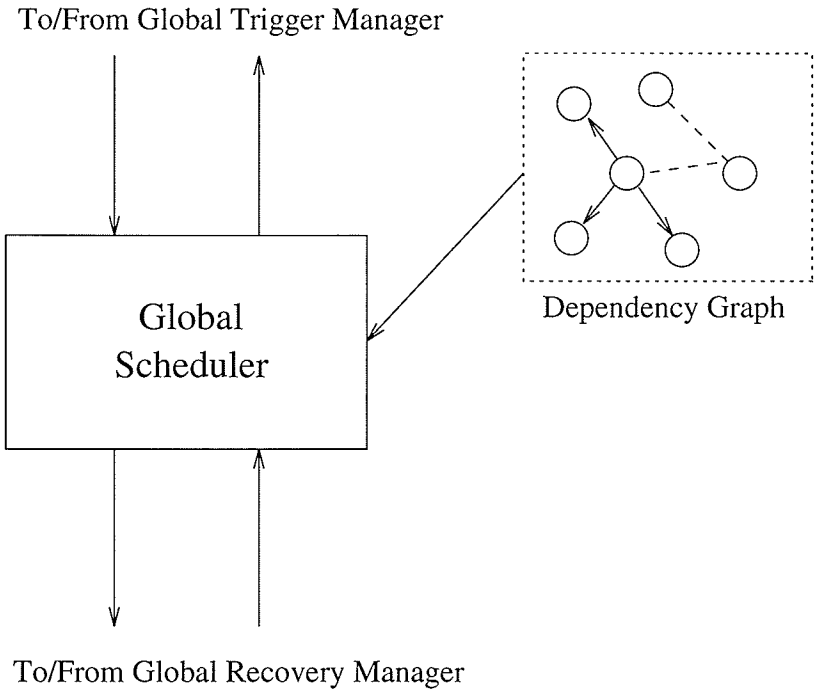

Figure 5.1: Global Scheduler

Example 5.2.1 Consider Example 5.1.1. GT_1 and $GT_{1.1}$ belong to the same global transaction family. If our algorithm schedules with respect to global transaction *families* instead of global transactions then we have the following:

$$\begin{array}{lllll} GTF_1\text{:} & GST^1_{1(1)} & GST^2_{1.1(1)} & & \\ GTF_2\text{:} & & GST^2_{2(2)} & GST^3_{2(2)} & \\ GTF_3\text{:} & GST^1_{3(3)} & & GST^3_{3(3)} & \end{array}$$

Suppose that GTF_1 is submitted first, followed by GTF_2. Both are scheduled accordingly. GTF_3 is the last to be submitted. Our modified algorithm will block GTF_3 so as to avoid a possible family order discrepancy. ■

Example 5.2.2 Consider Example 5.1.2. Our modified algorithm must examine the dependency graph created by the global trigger manager to ensure causal dependencies are not violated. Assume GTF_1 arrives at the global scheduler first and is submitted. GTF_2 arrives next and is submitted. $GTF_{1.1}$ is the last to arrive but is blocked because it overlaps with GTF_2 which overlaps with GTF_1 on which $GTF_{1.1}$ is causally dependent. Blocking $GTF_{1.1}$ ensures that a causal discrepancy will not occur. ■

Through examination of Examples 5.2.1 and 5.2.2 two ideas for the AGS present themselves:

1. Schedule global transaction families instead of global transactions. This will prevent family order discrepancies.

2. Block GSTs whose candidate sets contain one or more GTFs on which they are causally dependent. The DG created by the GTRM is used to establish any direct or indirect causal discrepancies.

Our AGS must incorporate these two ideas in order to schedule AMDB-serializable histories.

Active Global Scheduler (*Initial GTF Submission*)

1. Each global subtransaction is scheduled independently. Each is submitted or suspended based on the activities of other active global transaction families. All global subtransactions form a unique set of candidate GTFs which may cause the GST to be suspended. The candidate set is based on the DBMS to which the GST is to be submitted. For example, given $GST^k_{p(i)} \epsilon GTF_i$ to be submitted to $DBMS^k$, the candidate set is composed of all global transaction families which are waiting to access data at LDB^k or which have an active GST at $DBMS^k$. In addition to these GTFs, any GTF which overlaps with them may cause non-AMDB-serializable schedules. The entire set is considered when attempting to submit a global subtransaction.

   Given an arbitrary GTF_j in the candidate set, if the intersection of GTF_j's DBMS set and the set of other DBMSs to be accessed by GTF_i is not empty, then $GST^k_{p(i)}$ cannot be submitted so Step (3) is executed. Otherwise, Step (2) is performed.

2. If there exists a GT_t in the candidate set which GTF_j is causally dependent on then $GST^k_{p(i)}$ cannot be submitted so Step (3) is executed. Otherwise, $GST^k_{p(i)}$ is submitted. If there is another global subtransaction to be submitted, it is tested at Step (1). Otherwise, the algorithm terminates.

3. $GST^k_{p(i)}$ cannot be submitted immediately, so it is suspended in a wait queue and the next subtransaction is tested at Step (1). If no other GSTs need to be tested the algorithm terminates.

Active Global Scheduler (*GSTs Completion Process*)

1. The completion of the global subtransaction is recorded.

2. Completion of a global subtransaction may permit the submission of waiting subtransactions. This involves testing every waiting global subtransaction at the DBMS where the completing GST was active.

3. The global subtransaction at the head of the wait queue for the DBMS undergoes the same testing procedure as in Steps (1) and (2) of the initial global transaction family arrival process described above. If a GTF is active which can cause a non-AMDB-serializable schedule, Step (4) is performed, otherwise Step (5).

4. Since a GTF is still present that could cause a non-AMDBSR schedule the GST remains passive. The next GST waiting is tested at Step (3) unless the entire queue has been tested which terminates the process.

5. The GST can be submitted so its status is changed from passive to active. If GSTs are waiting, each must be retested at Step (3) or the algorithm terminates.

The following is a list of routines/data structures required by this algorithm:

DBMS_set(GTF_i) : The set of DBMSs where GTF_i submits subtransactions.

Active_set($DBMS^k$) : The set of global transaction families which have an active global subtransaction executing at $DBMS^k$.

card : Cardinality function which returns the number of elements in the argument set.

Algorithm 5.1 *(Active Global Scheduler - Initial Scheduler)*

begin

input GTF_i : global transaction family;

var candidate_set : **set of** GTF identifiers;

DBMS_set(GTF_i) $\leftarrow$ **set of** DBMSs accessed; (1

GSTs_active(GTF_i) $\leftarrow \phi$; (2

GSTs_passive(GTF_i) $\leftarrow \phi$; (3

GSTs_complete(GTF_i) $\leftarrow \phi$; (4

for each $GST^k_{p(i)} \epsilon$ DBMS_set(GTF_i) **do** (5

begin

candidate_set $\leftarrow$ Active_set($DBMS^k$) $\bigcup$ Wait_Q($DBMS^k$) - GTF_i; (6

for each $GTF_m \epsilon$ candidate_set **do** (7

if $\exists$ active GTF_n ($m \neq n$, $i \neq n$) such that

DBMS_set(GTF_m) $\bigcap$ DBMS_set(GTF_n) $\neq \phi$ **then** (8

candidate_set $\leftarrow$ candidate_set $\bigcup GTF_n$; (9

if $\exists GTF_j \epsilon$ candidate_set such that (card($\mathcal{GST}_j$) > 1 **and** card($\mathcal{GST}_i$) > 1 **and**

DBMS_set(GTF_j) $\bigcap$ (DBMS_set(GTF_i) - $DBMS^k$) $\neq \phi$)

or ($\exists GT_t \epsilon$ candidate_set with GTF_i causally dependent on GT_t) **then** (1

begin

GSTs_passive(GTF_i) $\leftarrow$ GSTs_passive(GTF_i) $\bigcup$ $DBMS^k$; (1

passivate $GST^k_{p(i)}$ **on** Wait_Q($DBMS^k$); (1

end (* if *)

else begin

GSTs_active(GTF_i) $\leftarrow$ GSTs_active(GTF_i) $\bigcup$ $DBMS^k$; (1

Active_set($DBMS^k$) $\leftarrow$ Active_set($DBMS^k$) $\bigcup GTF_i$; (1

submit $GST^k_{p(i)}$ **to** $DBMS^k$; (1

end; (* else *)

end; (* for *)

end;

Algorithm 5.2 *(Active Global Scheduler - Subtransaction Termination)*

begin

input $GST^k_{p(i)}$: GST for GTF_i at $DBMS^k$ completes;

var candidate_set : **set of** GT identifiers;

Active_set($DBMS^k$) $\leftarrow$ Active_set($DBMS^k$) - GTF_i; (1)

GSTs_active(GTF_i) $\leftarrow$ GSTs_active(GTF_i) - $DBMS^k$; (2)

GSTs_complete(GTF_i) $\leftarrow$ GSTs_complete(GTF_i) $\bigcup$ $DBMS^k$; (3)

for each $GST^k_{s(j)} \epsilon$ Wait_Q($DBMS^k$) **do** (4)

begin

candidate_set $\leftarrow$ Active_set($DBMS^k$) $\bigcup$ ($\{GTF_l \mid GTF_l \epsilon$ Wait_Q($DBMS^k$)

$\wedge$ GTF_l is active$\}$ - GTF_j); (5)

for each $GTF_m \epsilon$ candidate_set **do** (6)

if $\exists$ active GTF_n ($m \neq n$, $j \neq n$) such that

DBMS_set(GTF_m) $\bigcap$ DBMS_set(GTF_n) $\neq \phi$ **then** (7)

candidate_set $\leftarrow$ candidate_set $\bigcup GTF_n$; (8)

if $\exists GTF_r \epsilon$ candidate_set such that (card($\mathcal{GST}_r$) > 1 **and** card($\mathcal{GST}_j$) > 1 **and**

DBMS_set(GTF_r) $\bigcap$ (DBMS_set(GTF_j) - $DBMS^k$) $\neq \phi$)

or ($\exists GT_t \epsilon$ candidate_set with GTF_j causally dependent on GT_t) **then** (9)

repassivate $GST^k_{s(j)}$ **on** Wait_Q($DBMS^k$) (10)

else begin

GSTs_passive(GTF_j) $\leftarrow$ GSTs_passive(GTF_j) - $DBMS^k$; (11)

GSTs_active(GTF_j) $\leftarrow$ GSTs_active(GTF_j) $\bigcup$ $DBMS^k$; (12)

Active_set($DBMS^k$) $\leftarrow$ Active_set($DBMS^k$) $\bigcup GTF_j$; (13)

remove $GST^k_{s(j)}$ **from** Wait_Q($DBMS^k$); (14)

submit $GST^k_{s(j)}$ **to** $DBMS^k$; (15)

end; (* else *)

end; (* for *)

end;

Figure 5.3: Active Global Scheduler (Part 2)

Wait_Q($DBMS$) : Global subtransactions which cannot be submitted immediately are placed on a wait queue for access to the DBMS. There is one Wait_Q for each DBMS.

GSTs_active(GTF_i) : The set of DBMSs which have an active subtransaction of GTF_i.

GSTs_passive(GTF_i) : The set of DBMSs which have a waiting subtransaction of GTF_i.

GSTs_active(GTF_i) : The set of DBMSs which have a complete subtransaction of GTF_i.

Figure 5.2 and 5.3 present the algorithm in two parts. The algorithm works similar to Barker's AGSS [6] but contains additional functionality to handle causal and family order discrepancies.

Figure 5.2 describes the GT initial submission process. Line (1) forms a set describing the DBMSs accessed by the GT. Line (2), (3), and (4) initialize the GTFs reference sets to record the status of their GSTs.

Global transaction families are tested to determine if an overlap exists with other GTFs. Each GST (line (5)) is tested for submission independently. The set of global transaction families which could cause the passivation of a global subtransaction are those that have a GST at or waiting to be submitted to that DBMS (line (6)). In addition to these GTFs, any others which overlap with them must be included in the candidate set (lines (7-9)). Assume that $GST^k_{p(i)} \epsilon GTF_i$ is to be submitted to $DBMS^k$. Line (10) determines if there exists a $GTF_j \epsilon$ candidate_set which accesses another DBMS accessed by GTF_i. If such a GTF_j exists, $GST^k_{p(i)}$ is passivated (lines (11-12)). Line (10) also determines if there exists a $GT_t \epsilon$ candidate_set on which

GTF_i is causally dependent. If so, $GST^k_{p(i)}$ is passivated (line (11-12)). If $GST^k_{p(i)}$ is not passivated as a result of line (10) then it may be submitted (lines (13-15)).

When a global subtransaction completes, the sets are updated to reflect the completion at lines (1-3) of Figure 5.3. Line (4) retests all waiting GSTs at this DBMS for submission eligibility. The condition for submission is nearly the same as when the GTF initially arrived. The candidate set is formed at line (5), but it is necessary to remove the GTF which we are attempting to submit the subtransaction for and any which are inactive. The rest of the algorithm is logically identical to the initial submission process.

The following provides a demonstration of the AGS.

Example 5.2.3 We extend Example 4.3.1 to include two additional GTs:

$$\begin{array}{lllll} GT_2: & GST_2^4 & & & GST_2^6 \\ GT_3: & & GST_3^5 & GST_3^6 & \end{array}$$

Two new global triggers GTR_4 ($Tdep$) and GTR_5 ($Cdep$) are also added and each fires once. This generates two global transactions each accessing only one database:

$$\begin{array}{lll} GT_{4.1}: & GST_{4.1}^5 & \\ GT_{5.1}: & & GST_{5.1}^6 \end{array}$$

$GT_{4.1}$ is triggered by GT_3 and $GT_{5.1}$ by $GT_{4.1}$. Finally, we add a sixth local database to the multidatabase of Example 4.3.1. The dependency graph of Figure 5.4 illustrates the various dependencies of our example global transactions.

The Active Global Scheduler (AGS) algorithm is demonstrated by the following execution sequence:

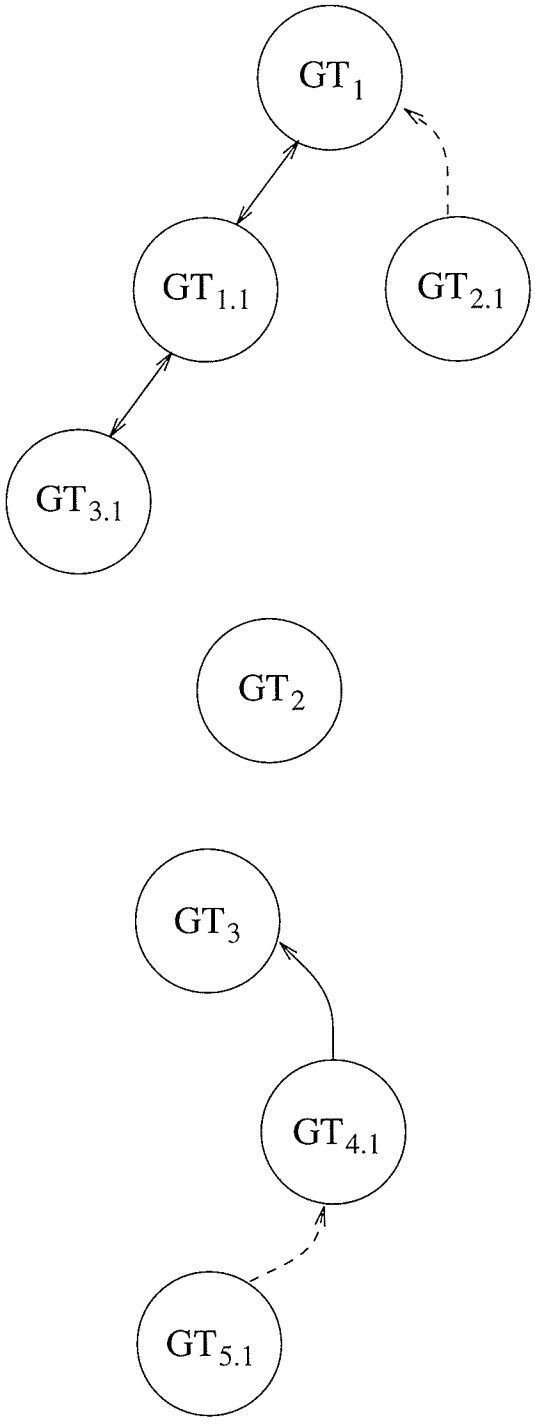

Figure 5.4: Global Transaction Dependencies of Example 5.2.3

1. GTF_1 (GT_1, $GT_{1.1}$, and $GT_{3.1}$) is presented to the AGS and the required sets are formed. Since no other GSTs are present at any DBMS all of the GSTs are submitted. The active sets of $DBMS^1$, $DBMS^2$, $DBMS^3$, $DBMS^4$, and $DBMS^5$ are updated. The GSTs_active(GTF_1) set is {$DBMS^1$,$DBMS^2$,$DBMS^3$, $DBMS^4$,$DBMS^5$}. Line (1) of Figures 5.5 and 5.6 illustrates this scenario.

2. GTF_2 arrives and each GST is tested for submissability. GST_2^4 is tested first (line (5)). The candidate set contains GTF_1 (lines (6-9)). The test for conflict fails (line (10)) and GST_2^4 is submitted (lines (20-22)). GST_2^6 is tested next and since there are no GSTs at $DBMS^6$ the candidate set is empty. The test for conflict fails again (line (10)) and GST_2^6 is submitted. The GSTs_active(GTF_2) set is {$DBMS^4$,$DBMS^6$}. Line (2) of Figures 5.5 and 5.6 depicts the situation.

3. $GT_{1.1}$ fires but is part of GTF_1 so it is handled as part of that family. $GT_{2.1}$ fires but is not part of the triggering family (it is only causally dependent). $GTF_{2.1}$ arrives at the scheduler and $GST_{2.1}^1$ is tested first. The candidate set is created (lines (6-9)) and is {GTF_1}. Consider the possible submission of $GST_{2.1}^1$ to $DBMS^1$:

   test for GTF_1 at $DBMS^1$

   $$= \text{DBMS\_set}(GTF_1) \cap (\text{DBMS\_set}(GTF_{2.1}) - DBMS^1)$$
   $$= \{DBMS^1,DBMS^2,DBMS^3,DBMS^4,DBMS^5\} \cap (\{DBMS^1,DBMS^5\} - DBMS^1)$$
   $$= \{DBMS^5\}$$

   This indicates that an overlap exists with an active transaction family of the candidate set. $GST_{2.1}^1$ must be placed on the Wait_Q($DBMS^1$) and GSTs_passive($GTF_{2.1}$)={$DBMS^1$}.

$GST^5_{2.1}$ is tested next. The candidate set is $\{GTF_1\}$. The following test demonstrates that $GST^5_{2.1}$ must be placed on Wait_Q($DBMS^5$):

test for GTF_1 at $DBMS^1$

$$= \text{DBMS\_set}(GTF_1) \bigcap (\text{DBMS\_set}(GTF_{2.1}) \text{ - } DBMS^5)$$
$$= \{DBMS^1, DBMS^2, DBMS^3, DBMS^4, DBMS^5\} \bigcap (\{DBMS^1, DBMS^5\} \text{ - } DBMS^5)$$
$$= \{DBMS^1\}$$

4. GTF_3 undergoes the same testing procedure. GST^5_3 is tested first and the candidate set is $\{GTF_{2.1}, GTF_1, GTF_2\}$.

test for $GTF_{2.1}$ at $DBMS^5$

$$= \text{DBMS\_set}(GTF_{2.1}) \bigcap (\text{DBMS\_set}(GTF_3) \text{ - } DBMS^5)$$
$$= \{DBMS^1, DBMS^5\} \bigcap (\{DBMS^5, DBMS^6\} \text{ - } DBMS^5)$$
$$= \phi$$

test for GTF_1 at $DBMS^5$

$$= \text{DBMS\_set}(GTF_1) \bigcap (\text{DBMS\_set}(GTF_3) \text{ - } DBMS^5)$$
$$= \{DBMS^1, DBMS^2, DBMS^3, DBMS^4, DBMS^5\} \bigcap (\{DBMS^5, DBMS^6\} \text{ - } DBMS^5)$$
$$= \phi$$

test for GTF_2 at $DBMS^5$

$$= \text{DBMS\_set}(GTF_2) \bigcap (\text{DBMS\_set}(GTF_3) \text{ - } DBMS^5)$$
$$= \{DBMS^4, DBMS^6\} \bigcap (\{DBMS^5, DBMS^6\} \text{ - } DBMS^5)$$
$$= \{DBMS^6\}$$

Since the test succeeds in the third case, the subtransaction must be passivated to ensure correct AMDB-serializability. The second GST is tested in the same fashion and must be passivated. See line (4) of Figures 5.5 and 5.6. Note that Barker's AGSS would allow GST_3^5 and GST_3^6 to be submitted. However, this could cause a family order discrepancy with $GT_1 \prec GT_2 \prec GT_3 \prec GT_{3.1}$.

5. Next, GST_1^1 completes at $DBMS^1$. The active set for $DBMS^1$, active set for GTF_1, and the complete set for GTF_1 are updated (lines (1-3) of Figure 5.2). $GST_{2.1}^1$ is currently in the wait queue for $DBMS^1$ and is retested. The candidate set is empty so the test (line (9)) fails and $GST_{2.1}^1$ is submitted (lines (11-15)). Line (5) of Figures 5.5 and 5.6 illustrates the active set for each DBMS.

6. $GT_{3.1}$ fires next but is part of GTF_1 so scheduler operations are unaffected. GST_1^4 completes and GST_2^4 is submitted. GST_2^6 also completes and GST_3^6 is submitted (see line (6) of Figures 5.5 and 5.6).

7. $GT_{4.1}$ fires and $GST_{4.1}^5$ is its only subtransaction. The candidate set is $\{GTF_1$, $GTF_{2.1}$,GTF_3,$GTF_2\}$. There is only one GST so the first part of the test (line (10)) fails. However, $GT_3 \in$ candidate_set and $GTF_{4.1}$ is causally dependent on GT_3, so the second half of the test succeeds and $GST_{4.1}$ is passivated on the wait queue (lines (11-12)) (line (7) of Figures 5.5 and 5.6). Note that the AGSS [6] would submit $GST_{4.1}^5$.

8. Next, $GST_{2.1}^1$, GST_1^2, and $GST_{3.1}^5$ all complete their executions. The wait queues for $DBMS^1$ and $DBMS^2$ are empty so checks are not required for those. There are no GSTs currently active at $DBMS^5$ so $GST_{2.1}^5$ is immediately submitted from the wait queue. GST_3^5 is also tested for submissability. The candidate set is $\{GTF_2$,GTF_1,$GTF_{2.1}\}$.

test for GTF_2 at $DBMS^5$

$$= \text{DBMS\_set}(GTF_2) \cap (\text{DBMS\_set}(GTF_3) - DBMS^5)$$
$$= \{DBMS^4, DBMS^6\} \cap (\{DBMS^5, DBMS^6\} - DBMS^5)$$
$$= \{DBMS^6\}$$

The test indicates that GST_3^5 should remain in the wait queue. $GST_{4.1}^5$ is also tested but it is causally dependent on GTF_3 which is in the candidate set. $GST_{4.1}^5$ remains in the wait queue (line (8) of Figures 5.5 and 5.6).

9. $GST_{1.1}^3$ and $GST_{2.1}^5$ complete. GST_3^5 is submitted and $GST_{4.1}^5$ retested. $GST_{4.1}^5$ is causally dependent on GTF_3 and so is repassivated on the wait queue (line (9) of Figures 5.5 and 5.6).

10. GST_3^5 completes and $GST_{4.1}^5$ is submitted. $GT_{5.1}$ fires and contains $GST_{5.1}^6$ as its only subtransaction. The candidate set is $\{GTF_3, GTF_{4.1}, GTF_2\}$. The first part of the test (line (10)) fails as $\text{card}(\mathcal{GST}_{5.1}) = 1$. However, the second part succeeds as $GTF_{5.1}$ is causally dependent (transitively) on GT_3. Hence, $GST_{5.1}^6$ is passivated on the wait queue (line (10) of Figures 5.5 and 5.6).

11. Finally, GST_2^4, $GST_{4.1}^5$, and GST_3^6 complete and $GST_{5.1}^6$ is submitted and completes (line (11) of Figures 5.5 and 5.6). ■

5.2.1 Correctness of the Active Global Scheduler

We will use the same two step procedure described in Barker [6]. First, we will identify some attributes of histories produced by the AGS. Second, we provide a theorem which proves that the AGS produces only AMDB-serializable histories.

	Active_set($DBMS^1$)	Active_set($DBMS^2$)	Active_set($DBMS^3$)
(0)	ϕ	ϕ	ϕ
(1)	$\{GTF_1\}$	$\{GTF_1\}$	$\{GTF_1\}$
(2)	$\{GTF_1\}$	$\{GTF_1\}$	$\{GTF_1\}$
(3)	$\{GTF_1\}$	$\{GTF_1\}$	$\{GTF_1\}$
(4)	$\{GTF_1\}$	$\{GTF_1\}$	$\{GTF_1\}$
(5)	$\{GTF_{2.1}\}$	$\{GTF_1\}$	$\{GTF_1\}$
(6)	$\{GTF_{2.1}\}$	$\{GTF_1\}$	$\{GTF_1\}$
(7)	$\{GTF_{2.1}\}$	$\{GTF_1\}$	$\{GTF_1\}$
(8)	ϕ	ϕ	$\{GTF_1\}$
(9)	ϕ	ϕ	ϕ
(10)	ϕ	ϕ	ϕ
(11)	ϕ	ϕ	ϕ

Figure 5.5: Trace of the AGS execution (part 1)

	Active_set($DBMS^4$)	Active_set($DBMS^5$)	Active_set($DBMS^6$)
(0)	ϕ	ϕ	ϕ
(1)	$\{GTF_1\}$	$\{GTF_1\}$	ϕ
(2)	$\{GTF_1, GTF_2\}$	$\{GTF_1\}$	$\{GTF_2\}$
(3)	$\{GTF_1, GTF_2\}$	$\{GTF_1\}$	$\{GTF_2\}$
(4)	$\{GTF_1, GTF_2\}$	$\{GTF_1\}$	$\{GTF_2\}$
(5)	$\{GTF_1, GTF_2\}$	$\{GTF_1\}$	$\{GTF_2\}$
(6)	$\{GTF_2\}$	$\{GTF_1\}$	$\{GTF_3\}$
(7)	$\{GTF_2\}$	$\{GTF_1\}$	$\{GTF_3\}$
(8)	$\{GTF_2\}$	$\{GTF_{2.1}, GTF_3\}$	$\{GTF_3\}$
(9)	$\{GTF_2\}$	$\{GTF_{4.1}\}$	$\{GTF_3\}$
(10)	$\{GTF_2\}$	$\{GTF_{4.1}\}$	$\{GTF_3\}$
(11)	ϕ	ϕ	ϕ

Figure 5.6: Trace of AGS execution (part 2)

In Chapter 3 we assume that each DBMS is capable of correctly serializing all submitted transactions. This is stated in the following proposition.

Proposition 5.2.1 Each local scheduler always schedules all transactions in a serializable order. ■

As described in Chapter 4 the global trigger manager embeds GSTs submitted to the same site if their GTs belong to the same global transaction family. The GTRM uses the same order for embedding for each DBMS so that it does not violate serializability. The embedding order for a GTF is equal to the order constructed by an ordered traversal of the family tree. A result of this is specified in the following Proposition and is used in the proof of Theorem 5.2.1.

Proposition 5.2.2 If $GT_{j(p)}$ is a descendent of $GT_{i(p)}$ and $GT_{k(p)}$ is not then $GT_{k(p)}$'s embedding order is either before both $GT_{i(p)}$ and $GT_{j(p)}$ or after both. ■

The following lemma is presented in Barker [6] and is used to prove the GSS and AGSS algorithms correct. We present it here to support the proof of the AGS.

Lemma 5.2.1 For every pair of global transactions GT_i, $GT_j \epsilon \mathcal{GT}$, scheduled by the AGS, either all of GT_i's subtransactions are executed before GT_j's at every DBMS or vice versa.

Proof: The AGS and AGSS are similar algorithms so the proof described in Lemma 4.3 of Barker (page 68) [6] is modified for our purposes. First we will prove that for every pair of families GTF_i, GTF_j $\epsilon \mathcal{GTF}$, either all of GTF_i's subtransactions are executed before GTF_j's at every DBMS or vice versa.

Consider GTF_a requiring only a single subtransaction (eg. $GST^k_{i(a)}$). From Proposition 5.2.1 it follows that any other $GST^k_{j(b)} \epsilon GTF_b$ either precedes or follows $GST^k_{i(a)}$.

Global transaction families which submit to multiple databases but only access one database in common with all other active GTFs are submitted immediately. This occurs because each GST is tested independently (line(5) Figure 5.2). Since only one DBMS is accessed in common for each GTF in the candidate set (lines(6-9)) the test will fail at line(10) of Figure 5.2 which results in the submission of each GST. Since each GTF in the candidate set overlaps at only one DBMS, it follows without loss of generality that, when we consider a $GST^k_{i(a)} \epsilon GTF_a$ and $GST^k_{j(b)} \epsilon GTF_b$ whose base-sets overlap, any ordering chosen by $DBMS^k$ is consistent. That is, since $DBMS^k$ is the only place where GTF_a and GTF_b access common data, we know from Proposition 5.2.1 that either $GTF_a \prec GTF_b$ or $GTF_b \prec GTF_a$, as required.

Global transaction families that overlap at more than a single DBMS are always submitted and executed in the same order at each overlapping DBMS. Suppose we have two arbitrary active global subtransactions $GST^k_{j(a)}$, $GST^l_{j(a)} \epsilon GTF_a$ which access $DBMS^k$ and $DBMS^l$, respectively. Consider the attempt to submit $GST^k_{i(b)}$. Since there exists another GST at $DBMS^l$, either waiting or active, $GST^k_{i(b)}$ is passivated (lines (11-12) of Figure 5.2). $GST^l_{i(b)}$ is blocked for the same reason. Consider the completion of GTF_a. Assuming that there are no other similar GSTs, the test at line (9) of Figure 5.3 allows $GST^k_{i(b)}$ to be submitted. Since $GST^k_{j(a)}$ is complete, it follows immediately that $GST^k_{j(a)} \prec GST^k_{i(b)}$. $GST^l_{i(b)}$ is submitted in the same way. It follows that $GST^l_{j(a)} \prec GST^l_{i(b)}$. Clearly the GSTs of GTF_a

precede the GSTs of GTF_b at every DBMS, as required.

To complete the proof we must show that for every pair of transactions $GT_{i(a)}$, $GT_{j(a)} \epsilon GTF_a$ either all of GT_i's GSTs are executed before GT_j's at every DBMS or vice versa. $GT_{i(a)}$ and $GT_{j(a)}$ belong to the same GTF so if they access a common $DBMS^k$ then GST_i^k and GST_j^k are embedded in the same submitted GST. The embedding order used by the GTRM is the same at each DBMS where the GTs intersect. Hence, either $GT_i \prec GT_j$ or $GT_j \prec GT_i$ at every DBMS, which completes the proof. ■

We are now in a position to prove the AGS correct. This involves showing that all histories produced by the AGS are AMDB-serializable.

Theorem 5.2.1 The AGS produces only AMDB-serializable histories.

Proof: To prove that the AGS creates only AMDB-serializable schedules we must show that the AMSG of an arbitrary history produced by the AGS is $\Lambda - acyclic$, $\Gamma - acyclic$, *family order discrepancy free*, and *causal discrepancy free.*

Acyclic: To show that the AMSG is $\Lambda - acyclic$ and $\Gamma - acyclic$ we appeal to Theorem 4.1 of Barker [6]. This theorem states that if a scheduler algorithm can satisfy Proposition 5.2.1 and Lemma 5.2.1 then it produces histories with acyclic AMSGs. The AGS is one such scheduler so the acyclicity of the AMSGs is proven.

Family order discrepancy free: Next, we must show that the AMSG is family order discrepancy free. Assume that a family order discrepancy does exist. Then there exists a path $GT_{i(p)} \rightarrow ... \rightarrow GT_k \rightarrow ... \rightarrow GT_{j(p)}$ in the AMSG where GT_j is a child of GT_i and GT_k is not a descendent of GT_i. Two cases are possible:

Case 1: GT_k is not a member of GTF_p. Each arc into or out of a GT node represents a conflict between GSTs of the nodes at some site. If all of the conflicts occur

at the same DBMS then Proposition 5.2.1 is violated. Hence, at least one GT node (other than GT_i and GT_j) on the path conflicts at two different sites. Without loss of generality, let this be GT_k. Consider the submission of one of GT_k's conflicting GSTs. Since every GT on this path overlaps the candidate set would include every GTF on the path. Thus, the site of the other conflicting GST of GT_k would intersect with a candidate GTF and the GST would not be submitted. The same is true for the submission of the other GST. Hence, GT_k must be a member of GTF_p.

Case 2: GT_k is a member of GTF_p but is not a descendent of GT_i. From case (1) above we know every GT on the path is a member of GTF_p. Furthermore, we know that none of the nodes represent LTs. If there did exist a LT then Proposition 5.2.1 would be violated. As a result of GTRM family embedment we know that there is only one GST submitted to each site accessed by GTF_p. Since GT_i executes a conflicting operation before another family member GT_a, it must have been embedded before GT_a. This principle can be applied to GT_a and the family member GT_b that it conflicts with. Hence, GT_i is embedded before GT_b as well (if they access similar sites). If follows that GT_i is the "oldest" with respect to embedment, GT_j is the "youngest", and GT_k is between them. However, this violates Proposition 5.2.2. Hence, GT_k must be a descendent of GT_i.

Causal discrepancy free: Finally, we must show that the AMSG is causal discrepancy free. Assume the AMSG contains a causal discrepancy. That is, there is a $\Lambda, \Gamma - path$ from GT_j to GT_i and a $\pi - path$ from GT_i to GT_j

Suppose GT_i and GT_j are members of different families. Consider the submission of GT_j. GT_i caused GT_j to occur and so is active at one or more DBMSs when GT_j is received by the GS. There is a path from GT_j to GT_i so every GTF on this path would have been in the candidate_ set when GT_j was submitted. However, line (10) of Figure 5.2 and line (9) of Figure 5.3 would not submit GT_j's GSTs but

would block them until GT_is family completes. Thus, GT_i and GT_j must belong to the same family.

The path must contain some GT_k not in the family or else GT_j's embedment order would be earlier than GT_i's which could never occur (see case (2) of family order discrepancy free section). Suppose that the conflict arcs of GT_k occur at two different DBMSs. Every GT on the path would be in the candidate set when GT_k was submitted and GT_k's GSTs would be blocked (see case (1) of family order discrepancy free section). Hence, every GT not in the family and on the path conflicts at the same DBMS. This implies that operations of GT_j precede those of GT_k which precede those of GT_i at some DBMS. This violates Proposition 5.2.1. Therefore, the AMSG is causal discrepancy free and the theorem is proved. ■

Chapter 6

Conclusion

Trigger and transaction management issues in active multidatabase systems are studied in this thesis. This thesis has contributed in a number of important ways. Barker's formal model of multidatabase system is extended to a formal model of active multidatabase system. This paradigm uses global triggers to automatically perform database actions when certain conditions in the database arise. Global triggers have a coupling mode associated with them which indicates how the trigger is to be executed with respect to the triggering transaction.

Histories in this thesis are considered correct if they are *AMDB-serializable*. AMDB-serializability is an extension of Barker's MDB-serializability. The extensions include restrictions to ensure transaction families are ordered properly and that causally dependent transactions appear to have executed after their triggering transactions. Additionally, a new *active multidatabase serializability graph* (AMSG) is used to simplify the process of determining whether a multidatabase history is AMDB-serializable. An AMDB-serializability theorem is presented and its correctness proved.

Trigger execution and event detection mechanisms are presented in this thesis. A unique method of event detection is used where the global subtransactions signal when simple events of global triggers occur. The Global Trigger Manager receives these messages and decides which global triggers to execute. Special techniques for subtransaction embedment are used to overcome deadlock anomalies which may arise from the full dependency coupling mode. A major result of this research is the ability to detect complex events that span more than one local database, even if the databases are heterogeneous. All of this is accomplished without a violation of local autonomy.

This thesis also contributes by presenting a scheduler algorithm (AGS) which ensures AMDB-serializable executions. This algorithm is a modification of Barker's AGSS which ensures MDB-serializability. The algorithm works by scheduling at the global subtransaction level as opposed to the traditional read/write level. The Active Global Scheduler is proved correct by demonstrating that any arbitrary history produced by it is AMDB-serializable.

Many aspects of trigger and transaction management are not discussed in this thesis. Some of these open research problems are enumerated by Barker [6]. These include:

1. The formation of global subtransactions from a global transaction.

2. Consideration of heterogeneity in this research. Certain aspects of local databases may lend to an improvement in transaction management. Specifically, would it be possible to exploit a local system that already has built-in event detection.

3. The global scheduler could incorporate semantic information to increase concurrency. This would involve detecting the meaning of a subtransaction and

scheduling accordingly.

4. The full autonomy assumption could be relaxed. This would allow more modifications to the local databases that should reveal ways of improving concurrency and reliability. However, this may not always be possible.

There are also many open problems related to the original work presented in this thesis. Barker's [6] ideas for reliability and recovery will have to be extended to fit the work of this thesis. This includes adding functionality to the Global Recovery Manager so that it can cope with failures such as lost signals and queries or aborting subtransactions. Furthermore, the GRM will have to use the Dependency Graph (DG) to control commitment and abortion of Global Transactions. The GRM would have the job of purging the DG and the Event Log (EL) when transactions have completed.

A prototype of the algorithms should be developed so that performance studies can be made. The results may reveal any inefficiencies in this work so it can be optimized. This includes implementing the Global Trigger Manager algorithms as well as the Active Global Scheduler algorithm. These results can also be compared to the simulation results of Barker's work to get a "feel" for how much of an increase in overhead the active techniques introduce.

This work assumed that trigger events were restricted to read and write events or complex combinations of events such as conjunction and disjunction. We did not address the issue of temporal event detection in a multidatabase environment. An example temporal event is "@3:00pm" which occurs at 3 oclock every day. The fact that each DBMS has its own clock makes this a very difficult area of distributed research. Also, this thesis did not allow events such as program executions to fire triggers.

The coupling modes presented in this thesis provide a reasonable amount of flexibility in describing how a trigger should be executed. Additional coupling modes may be desired to achieve other types of trigger executions. This may lead to drastic changes in the algorithms presented depending on the nature of the modes created.

This thesis assumes that local database systems are initially not active. Future work could involve using the existing active components of a local database system for trigger management at the multidatabase level. This could improve efficiency of event detection. However, this may not be possible without a further violation of autonomy.

Some ideas of Arizio *et al.* [5] could be applied to this work. One in particular is the analysis of local transactions for possible fired triggers. If a local transaction could never fire a trigger then it remains a local transaction. However, if the local transaction could potentially fire one or more triggers then it would become a global transaction with one GST.

Finally, the results of this thesis should be considered with other multidatabase architectures and models. The result that local autonomy is not violated in this approach makes it worthy of attention.

6.1 Interdatabase Dependencies

The model and algorithms described in this thesis are important research in the field of multidatabase systems. Maintaining interdependent data in these systems in a way such that the autonomy of the local systems is not violated is a difficult and open problem. The ideas of this thesis could be used to solve that problem.

Consider Example 1.1.1 from Chapter 1. A multidatabase system consisting of a phone company database and a construction company database has two inter-database dependencies:

$$Project\_X\_copy1 == Project\_X\_copy2$$
$$Project\_X\_cost \leq Project\_Y\_cost$$

Using our model, triggers are required to handle these constraints. The following triggers are used for the first constraint:

Global Trigger 1 = Action: update Project_X_copy2 with new value
Event: update to Project_X_copy1
Coupling Mode: Full Dependency

Global Trigger 2 = Action: update Project_X_copy1 with new value
Event: update to Project_X_copy2
Coupling Mode: Full Dependency

When an update occurs to either copy one of the triggers will fire to update the other copy. The second constraint requires only one trigger:

Global Trigger 3 = Action: if (new value > Project_Y_cost) then abort
Event: update to Project_X_cost
Coupling Mode: Full Dependency

An update to Project_X_cost will cause the action to be executed in a global transaction. The action checks if the new value exceeds the cost of project Y and if so the transaction aborts. The full dependency coupling mode is used so the abort will cause the updating transaction to abort.

Special higher level constructs could be set up for common types of interdatabase dependencies to ease the complexity of creating multiple triggers to handle one task. The equality constraint is the most common type of dependency in multidatabase systems. An example of such is the first constraint of Example 1.1.1. Our higher level construct could be: equality_dependency(Project_X_copy1, Project_X_copy2). The GTRM would interpret this correctly and setup and submit the appropriate update transactions whenever one copy is updated.

Other types of dependencies that could be handled include aggregate, referential, and existence dependencies. For example, suppose the dependency C=average(D_i) exists on the multidatabase. Triggers could be setup to fire whenever a D_i is updated. The action of these triggers would update C with the correct average of the D_is.

Referential integrity is also very important. Suppose emp_1 is a data item on some local database which contains employee information regarding one employee. On another database exist several payroll records for this employee. If a user deletes the emp_1 record then the payroll records will exist without an employee record. Triggers could be used to delete the payroll records (or take some other action) when the emp_1 record is deleted.

Currently this model can only support *immediate consistency*. That is, interdatabase dependencies are always satisfied. However, often it is not necessary to be so severe. It may be acceptable to let dependent data become inconsistent and only maintain the constraint periodically. This type of consistency is called *eventual consistency*. For example, the first constraint of Example 1.1.1 could be enforced once every two hours or once every five updates. The copy may be slightly out of date but is still acceptable for some purposes. The GTRM would require some modifications so it could "delay" triggers or track the number of times an event

occurs on an inter-transaction basis.

Many of the dependency ideas discussed require the ability for transactions to communicate with each other. This is necessary for even simple constraints such as $Project\_X\_copy1 == Project\_X\_copy2$. To maintain this constraint the update trigger must know the value that is written to the first copy so that it may correctly update the second copy. The message sending mechanisms of Chapter 4 could be modified to handle these situations but this is beyond the scope of this thesis.

The interdatabase dependency problem is a very important one in multidatabase systems. Much research is required to adequately cope with this problem.

Bibliography

[1] R. Agrawal and N. Gehani. ODE (Object Database and Environment): The Language and the Data Model. In *ACM SIGMOD Conference on Management of Data*, pages 36–45, 1989.

[2] A. Aiken, J. Widom, and J. Hellerstein. Behaviour of Database Production Rules: Termination, Confluence, and Observable Determinism. In *ACM SIGMOD Conference on Management of Data*, pages 59–68, 1992.

[3] R. Alonso, D. Barbara, and S. Cohn. Data Sharing in a Large Heterogeneous Environment. In *Proceedings from the Seventh International Conference on Data Engineering*, pages 305–313, April 1991.

[4] R. Alonso, D. Barbara, and H. Garcia-Molina. Data Caching Issues in an Information Retrieval System. In *ACM-TODS*, pages 359–384, September 1990.

[5] R. Arizio, E. Bomitali, M. Demarie, A. Limoniello, and P. Mussa. Managing Inter-database Dependencies with Rules + Quasi-transactions. In *Proceedings from RIDE-IMS*, pages 34–41, April 1993.

[6] K. Barker. *Transaction Management on Multidatabase Systems.* PhD thesis, University of Alberta, 1990.

[7] C. Beeri and T. Milo. A Model for Active Object Oriented Database. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 337–349, September 1991.

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

[9] T. Bloom. Issues in the Design of Object-Oriented Database Programming Languages. In *OOPSLA Conference Proceedings*, pages 441–451, 1987.

[10] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating Object-Oriented Data Modeling with a Rule-Based Programming Paradigm. In *ACM SIGMOD Conference on Management of Data*, pages 225–236, 1990.

[11] S. Ceri. A Declarative Approach to Active Databases. In *Proceedings of the IEEE Conference on the Management of Data*, pages 452–456, 1992.

[12] S. Chakravarthy. Rule Management and Evaluation: An Active DBMS prospective. *SIGMOD RECORD*, 18(3):20–28, 1989.

[13] A. Cornelio and S. Navathe. Using Active Database Techniques For Real Time Engineering Applications. In *Proceedings of the IEEE Conference on the Management of Data*, pages 100–107, 1993.

[14] U. Dayal, B. Blaustein, and A. Buchmann. The HiPAC Project: Combining Active Databases and Timing Constraints. In *SIGMOD Record*, March 1988.

[15] U. Dayal, A. Buchmann, and D. McCarthy. Rules Are Objects Too: A Knowledge Model For An Active, Object-Oriented Database System. In *Proceedings of the 2nd International Workshop on Object-Oriented Database Systems*, pages 129–143, 1988.

[16] O. Diaz, N. Paton, and P. Gray. Rule Management in Object Oriented Databases: A Uniform Approach. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 317–326, September 1991.

[17] K. Dittrich, A. Kotz, and J. Mulle. An Event/Trigger Mechanism to Enforce Complex Consistency Constraints in Design Databases. *SIGMOD RECORD*, 15(3):22–36, September 1986.

[18] S. Gatziu and K. Dittrich. Detecting Composite Events in Active Database Systems Using Petri Nets. In *Proceedings from RIDE-ADS*, pages 2–9, February 1994.

[19] N. Gehani and H. Jagadish. Ode as an Active Database: Constraints and Triggers. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 327–336, September 1991.

[20] N. Gehani, H. Jagadish, and O. Shmueli. Event Specification in an Active Object-Oriented Database. In *ACM SIGMOD Confernece on Management of Data*, pages 81–90, 1992.

[21] A. Gupta and J. Widom. Local Verification of Global Constraints in Distributed Databases. In *ACM SIGMOD Conference on Management of Data*, pages 49–58, 1993.

[22] E. Hansen. Rule Condition Testing and Action Execution in Ariel. In *ACM SIGMOD Conference on Management of Data*, pages 49–58, 1992.

[23] E. Hanson. An Initial Report on the Design of Ariel. *SIGMOD RECORD*, 18(3):12–19, 1989.

[24] M. Hsu, R. Ladin, and D. McCarthy. An Execution Model for Active Database Management System. In *SIGMOD Record*, March 1988.

[25] G. Karabatis and A. Sheth. Specifying Interdependent Data: A Case Study at Bellcore. Technical report, University of Houston, July 1992.

[26] D. McCarthy and U. Dayal. The Architecture of an Active Database Management System. In *ACM SIGMOD Conference on Management of Data*, pages 215–224, 1989.

[27] D. Mukhopadhyay and G. Thomas. Practical Approaches to Maintaining Referential Integrity in Multidatabase Systems. In *Proceedings from RIDE-IMS*, pages 42–49, April 1993.

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

[29] T. Risch. Monitoring Database Objects. In *Proceedings of the 15th International Conference on Very Large Data Bases*, pages 445–453, 1989.

[30] M. Rusinkiewicz, A. Sheth, and G. Karabatis. Specifying Interdatabase Dependencies in a Multidatabase Environment. Technical report, Bellcore, March 1991.

[31] A. Sheth and P. Krishnamurthy. Redundant Data Management in Bellcore and BCC Databases. Technical report, University of Houston, June 1989.

[32] A. Sheth, Y. Leu, and A. Elmagarmid. Maintaining Eventual Consistency of Interdependent Data in Multidatabase Systems. Technical report, Bellcore, June 1991.

[33] A. Sheth and M. Rusinkiewicz. Management of Interdependent Data: Specifying Dependency and Consistency Requirements. In *Proceedings of the Workshop on the Management of Replicated Data*, pages 34–49, November 1990.

[34] A. Sheth, M. Rusinkiewicz, and G. Karabatis. Using Polytransactions to Manage Interdependent Data. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 556–581. Morgan Kaufmann, 1990.

[35] M. Stonebraker. The Integration of Rule Systems and Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, 4(5):415–423, October 1992.

[36] M. Stonebraker, M. Hearst, and S. Potamianos. A Commentary on the POSTGRES Rules System. *SIGMOD RECORD*, 18(3):5–11, September 1989.

[37] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On Rules, Procedures, Caching and Views In Data Base Systems. In *ACM SIGMOD Conference on Management of Data*, pages 281–290, 1990.

[38] M. Stonebraker and G. Kemnitz. The Postgres Next-Generation Database Management System. *Communications of the ACM*, 34(10):78–92, October 1991.

[39] S. Urban and L. Delcambre. Constraint Analysis: A Tool For Explaining The Semantics of Complex Objects. In *Proceedings of the 2nd International Workshop on Object-Oriented Database Systems*, pages 156–161, 1988.

[40] L. Voort and A. Siebes. Termination and Confluence of Rule Execution. Technical report, CWI, January 1993.