Events in an
Active Object-Relational Database System

by

Salah-Eddine Machani

Linköping Studies in Science and Technology
Master's Thesis No: LiTH-IDA-Ex-9634
September 1996

Abstract

This report presents syntax, semantics and implementation of rule definitions in an Object Relational Active Database Management System, AMOS. Both, event-based and condition-based rules are considered. However, the main focus is on the event component of rules. The definition, the deletion, and the management of rule events are investigated. Events can be simple or composite and might be specified as updates on stored or derived functions. The rules are implemented based on the concept of function monitoring; events are compiled to active functions and an incremental change monitoring technique is used to detect changes.

Résumé

Ce rapport présente une syntaxe de définition de règles actives ainsi que sa sémantique et son implementation dans un Système de Gestion de Bases de Données Active Objet Relationnel, AMOS. Les règles sont soit à base d'événements ou à base de conditions. Cependant, un plus grand intérêt est porté sur la partie événement des règles. La définition, la détection et la gestion des événements sont discutés en details. Les événements peuvent être simples ou composés et peuvent être spécifiés comme des mises à jours sur des functions de bases ou sur des fonctions derivées. L'implementation des règles est basée sur le concept de gestion de fonctions; les événements sont compilés en fonctions actives et une technique de monitoring du changement incrementale est utilisée pour la détection des changements.

This work has been done as a collaboration between the University of Versailles and Linköping University as a part of ERASMUS program of the ACT-NET research network on active databases group and partially financed by the University of Pierre et Marie Curie (Paris 6).

Laboratoire PRiSM - Université de Versailles
45, avenue des Etats-Unis
78035 Versailles Cedex
France

Department of Computer and Information Science
Linköping University
S-584 83 Linköping
Sweden

Contents

Summary in French

Ce rapport présente un travail de recherche d'une durée de 6 mois effectué dans le Laboratoire de Génie Systèmes de Bases de Données (EDSLab[1], *Engineering Database System Laboratory*) à l'Université de Linköping en Suède. Ce travail a été réalisé dans le cadre des exigences nécessaires pour l'obtention du Diplôme d'Etudes Approfondies en Méthodes Informatiques Appliquées aux systèmes Industrielles (DEA MISI) au laboratoire de Recherche en Informatique, PRiSM[2] à l'Université de Versailles- France.

Le travail consiste dans un premier temps à définir une syntaxe conviviale et sa sémantique pour la spécification des règles *Evénement-Condition-Action* (règles ECA) dans un Système de Gestion de Base de Données (SGBD) Active Objet relationnel, nommé AMOS[FRS93] et dans un deuxieme temps d'implémenter des algorithmes efficaces pour la gestion et la detection des événements spécifiés et l'évaluation des conditions.
AMOS (Active Mediators Object system) [FRS93] est un prototype de recherche, classé dans la catégorie des SGBD *Objets relationnels* (OR). Son

1. Pour plus d'information sur EDSLab, utilisez l'URL:
   http://www.ida.liu.se/labs/edslab
2. Pour plus d'information sur PRiSM, utilisez l'URL:
   http://www.prism.uvsq.fr

architecture permet de localiser, chercher, combiner, et contrôler des données dans les systèmes d'information avec plusieurs stations connectées entre elles en utilisant des réseaux de communication rapides. Cette architecture utilise l'approche des médiateurs qui introduisent un niveau de logiciel intermédiaire entre les bases de données et leur utilisation dans des applications et par des utilisateurs. Ces médiateurs sont actifs puisqu'ils supportent les facilités des bases de données actives. Un prototype d'AMOS a été developpé a partir de la version WS-Iris à mémoire principale de IRIS[FA+89]. L'element central d'AMOS est AMOSQL, un language de requêtes objet-relationnel qui offre une interface de requêtes declarative permettant de définir, charger et manipuler la base de données. AMOSQL est dérivé de OSQL[Lyn91] qui est un langage fonctionnel, ayant ces racines dans DAPLEX[Shi81]. Le language de requête AMOSQL est en plus influencé par les efforts de standarisation comme SQL3[Mel95] et OQL[Cat94]. Comme dans OSQL, les requêtes AMOSQL sont compilées á des plans d'exécution dans un langage logique OO appellé Object-Log[LR92]. AMOSQL étend OSQL principalement avec des règles actives et un système plus riche en types et en fonctionalités de base de données multiples[FaR97].

Dans sa premiere version le langage de règles dans AMOS ne supporte que des règles de type CA (*Condition-Action*) où les événements entraînés sont calculés à partir des Conditions par le compilateur de règles. Ces règles sont déclenchées implicitement quand des données sont modifiées et les nouvelles données satisfont la Condition de la règle. Ce type de règles est généralement considéré plus déclaratif et peut être facilement programmé. La spécification des événements déclencheurs de règles comme une partie de la définition des règles permet de spécifier des actions différentes quand une condition donnée est satisfaite, dépendament de l'événement soulevé. L'intégration des événements dans AMOS a été l'objet du travail presenté dans ce rapport. Cette intégration nécessite la définition d'un langage de spécification de règles conforme aux fonctions d'AMOSQL et l'utilisation des méthodes efficaces pour la détection des événements et l'évaluation des conditions. Une nouvelle syntaxe permettant de définir des règles ECA (*Evenement-Condition-Action*) a été ainsi implémentée. La sémantique de cette syntaxe est "Quand l'Evénement spécifié dans la défintion de la règle est detecté, évalue la Condition. Si la Condition est vraie exécute l'Action". Cette syntaxe permet en outre de définir une variation de type de règles. L'omission de la partie Condition définit une règle de type *Evenement-Action* (règles EA), dans cette classe de règles, la Condition est considérée coome étant toujours vraie et l'Action est exécutée suite à la détection de l'événement. L'omission de la partie Evenement crée une règle de type CA, celle-ci a la même sémantique que les règles CA dans la version précédente d'AMOS.

Le modèle de données d'AMOS est un modèle objet relationnel. Dans ce modèle tout est objet, y compris les *fonctions*, les *types* et les *règles*. Ainsi les règles peuvent être créées et supprimées comme tout autre objet. Deux autres commandes sont utilisées pour activer ou desactiver une règle. Les événements peuvent être simples ou composés et seuls les événements liés à l'exécution des opérations de mise-à-jour sont considérées. Les événements utilisateurs et les événements temporels seront considérés dans un futur travail. Un événement dans AMOS peut être défini comme "Un changement d'état de la base de données à un instant donné" (l'instant de l'occurence de l'événement est enrigistré pour faciliter l'intégration des événements temporels). Les types d'événements pris en compte sont la création ou la suppression d'un objet, l'insertion d'une valeur dans une fonction, la suppression d'une valeur d'une fonction, la mise à jour de la valeur d'une fonction, et les combinaisons logiques des ces événements. Une syntaxe conviviale a été définie pour exprimer ces types d'événements. Les événements composés sont définis par les formes logiques de conjonction (l'ordre des événements n'est pas pris en compte) et de disjonction dans ce premier temps. L'inclusion de la forme de négation et l'ordonnancement des événements seront a considérer dans un travail futur.
Les règles ECA integrées dans AMOS sont basées sur le concept de gestion de fonctions. L'Evénement spécifié est transfomé en une fonction AMOSQL qui peut contenir des conjonctions et des disjonctions dans le cas des événements composées. La partie Condition est compilée en une requête AMOSQL qui peut contenir à son tour des conjonctions, des disjonctions ou des négations. La condition est vraie si le résultat de la requête est non vide. L'Action est transfomée en une procédure AMOSQL qui peut contenir n'importe quelle expression AMOSQL sauf *commit*. Les données peuvent être passées de l'événement à la condition et de la condition à l'action de chaque règle par l'utilisation des variables de requêttes partagées. L'exécution de la *fonction événement* renvoie les données entraînées. Ces données seront passées à la *fonction condition* comme parametres en arguments, ainsi la requête de la condition ne sera appliquée que sur l'ensemble des données modifiées. Ceci donne une évaluation correcte et efficace de la condition.
La fonction événement est définie en terme des *delta-relations*. Ces dernieres sont générée par le compilateur de règles pour chaque fonction de base ou dérivée réferenecée dans la définition de l'événement. Ces delta-relations

enrigistrent toutes les mises-à-jours effectuées sur les fonctions de base au cours de la transaction. Au moment du *commit*, une procedure basée sur la méthode de l'évaluation incrémentale est invoquée pour propager les changements vers les delta-relations attribuées aux fonctions derivées.
Un réseau de propagation a été modèlisé pour permettre une propagation plus efficaces des changements. Ce réseau est maintenu après chaque activation ou désactivation d'une règle en l'insérant ou la supprimant du réseau respectivement. L'algorithme de propagation est basé sur la méthode *breadth-first*. Les règles déclenchées après la détection des événements sont insérées dans une chaîne triée. Le triage des règles dans cette chaîne est basé sur l'*ordre de priorité* attribué par l'utilisateur à chaque règle active. Ainsi quand une règle est selectionée pour l'évaluation de la condition, elle est sélectionnée de sorte qu'aucune autre règle dans la chaîne n'ait un ordre de priorité plus grand. L'Action étant une procédure AMOSQL, son exécution peut soulever de nouveaux événements et le declenchement de nouvelles règles ou la même règle. Ceci peut causer une boucle infinie du processus. Une limite fixe contraignant le nombre de fois qu'une règle puisse être exécutée est mise au point pour resoudre ce problème de terminaison.

Ce rapport est constitué de deux parties principales. La première partie présente la syntaxe de définition des règles ECA dans AMOS et sa sémantique avec un plus grand interrêt sur le composant Evénement de ces règles. La deuxième partie discute de la façon permettant de detecter l'occurence des événements en utilisant la technique de l'évaluation Incrementale[Skö94, SR96, FSR93].

Chapitre 1 introduit le modèle objet-relationnel et le concept des systèmes de gestion de base de donnèes actives. Il donne aussi une vue génerale sur l'architecture du système AMOS et son langage de règles. Et enfin ce chapitre résume les motivations qui nous ont amenés à integrer les règles ECA dans AMOS et le travail effectué dans ce sens.

Chapitre 2 définit le modèle de données d'AMOS et son extension avec un langage de règles permettant de spécifier et de gérer des règles de type ECA.

Chapitre 3 présente le langage de spécification des événements. Un ensemble d'exemples est donné pour illustrer les différents types d'événements qui peuvent être spécifiés.

Chapitre 4 introduit la méthode de l'évaluation Incrementale et le concept des delta-relations et montre comment les fonctions générées à partir des événements spécifiées dans les définitions de règles sont définies en terme de ces delta-relations.

Chapitre 5 disctute l'algorithme de processus de règle et souligne l'algorithme utilisé pour implémenter la méthode de l'évaluation incrémentale .

Chapitre 6 conclut avec un résumé des principaux aspects de la nouvelle syntaxe de définition de règles actives dans AMOS et donne des perspectives pour des extensions de cette syntaxe dans le futur.

Preface

This report presents a significant work done on extending the rule language of an Object Relational Database Management System (ORDBMS), called AMOS[FRS93] with rules having ECA (*Event-Condition-Action*) rule semantics. The report is divided into two main parts. The first part describes a syntax for rule definitions and discusses its semantics with the focus on the event component. The second part investigates the detection and the management of simple and composite events using incremental evaluation techniques.

Outline

Chapter 1 introduces the object-relational model and the active database management systems. It also gives an overview of the AMOS architecture. This chapter also gives a summary of the work done in integrating ECA rules in AMOS and the motivations for this work. Related work is discussed at the end.

Chapter 2 defines the data model of AMOS and presents the new syntax for defining ECA rules and its semantics. A data base sample is given at the end to be referenced in the following chapters in illustrative examples.

Chapter 3 focuses on the event specification language. It presents the different types of events that can be specified. A set of examples are presented for illustration.

Chapter 4 introduces the delta relations approach and the incremetal evaluation techniques and shows how delta relations can be used to detect and record data modifications of stored relations and how changes to stored relations are propagated to derived relations using the incremental evaluation method. It then illustrates with examples how the event components in the rule definitions are compiled to active functions and how changes to these functions are monitored.

Chapter 5 explains some implementation aspects of rule management at the phases: rule creation, rule activation and deactivation, event detection and rule triggering, and rule execution at the check phase. It also shows the rule processing algorithm, the data structures of the propagation network, and the propagation algorithm.

Chapter 6 concludes with a summary of the main aspects of the new syntax and its semantics and presents some issues for future work.

Context

The work presented in this report has been carried on during a period of about 6 months at the Laboratory of Engineering Database Systems EDSLab[1] at Linköping University, Sweden and submitted to the Research Laboratory in Computer Science, PRiSM[2] in the University of Versailles, France as a partial fulfilment of the requirements for the Diploma of Advanced Studies (DEA in the French system is a 1 year preparatory course for a PhD) in the field of Industrial Applications of Computer Science.

Acknowledgements

I would like to thank my advisor, Professor Tore Risch, for giving me the opportunity to carry out this work and for his support and his excellent supervision. I would also like to thank Martin Sköld for providing me with rich ideas from his experience in the field and for helping me to get familiar with the AMOS system. Many thanks also for all EDSLab members for their fruitful discussions and suggestions

I am grateful to Anne Eskilsson for providing excellent administrative service.

Last, but not least, I owe special thanks to my professors Elisabeth Metais and Mokrane Bouzeghoub for their support and for helping me and encouraging me to join EDSLab.

Salah-Eddine Machani
Linköping, September 1996

1. For more information about EDSLab use URL: http://www.ida.liu.se/labs/edslab
2. For additional information about PRiSM use URL: http://www.prism.uvsq.fr

CHAPTER 1 Introduction

1.1 Object relational model

The DBMS market is still led by the relational database management systems (RDBMS), however the limitations of these systems when it comes to data modelling has led to the development of new database technology based on object oriented techniques and brought many researchers and industrials to investigate on these techniques.
Object-oriented databases improve relational systems by offering complex structures, object identity, inheritance between classes, and extensibility. The *first generation* OODBMS also usually includes basic database facilities such as a simple query language, access techniques such as hashing and clustering, transaction management, and concurrency control and recovery. However, they are incompatible with RDBMSs and do not include several RDBMS features such as a complete declarative query language, meta data management, views, and authorisation. Their advantage is a seamless integration with their corresponding OO programming language. Products originating from the first generation OODBMS approach are Gemstone, O2, Objectivity, ObjectStore, ONTOS, and Versant.
Systems called the *second generation* OODBMSs evolved from the classical relational database community and were also inspired by OO ideas. The attempt to meet the needs required by new types of database applications, as for instance from the scientific and engineering area, has resulted in an

extension of relational database technology with OO capabilities. Examples of these capabilities include object identity, object structure, composite objects, type constructors, encapsulation, inheritance, and OO extensions for relational query languages such as SQL-3[Mel95]. These DBMSs are called *Object Relational Database Management systems* (ORDBMs)[SM96]. Examples of this type of product are Odapter, Illustra, and UniSQL. The research prototype AMOS, that is used in this work, is based on this approach.

1.2 Active Databases

Traditional database management systems are passive: data is created, retrieved, modified, and deleted only in response to operations issued by users or application programs. Newer applications recognize the need for having a database system capable of reacting automatically in response to specific situations (to certain events occurring or to certain conditions being satisfied). Such systems are claimed to be active.
Active database behaviour is characterised by the definition of a set of ECA rules (Event-Condition-Action) as part of the database, which describe actions to be taken upon encountering an event in a particular database state. These rules are then associated with objects, making them responsive to a variety of events. Events range from data modification events (e.g., insert, delete, or update on a particular table in a relational database or a creation, deletion, modification of a particular object or a method invocation in an object-oriented database) to temporal events (e.g., 1 Sep. at 12:00, every day at 12:00, from 18:00:00 every 5 minutes) to application-defined events (e.g., user-login, mail reception). When the event is detected the relevant rules fire. Firing of a rule implies evaluating a, possibly complex, condition on the database, and carrying out the corresponding action. Conditions might be specified as database predicates, restricted predicates, database queries or application procedures. Actions may refer to a transaction (e.g., to abort it) and they may affect the database itself by performing some data modification operations or some data retrieval operations. An active database system derives its power from the variety of events it can respond to, how efficiently the condition is evaluated and the kind of actions it can perform in response.
Although the general and preferred form of active rules are ECA rules, other variations of active rules may occur: the omission of the condition part leads to an Event-Action rule (EA-rule) where the Condition is considered to be always true, and the omission of the event part leads to a Condition-Action rule (CA-rule) in which case the compiler or the ADBMS itself generate the

event definition.
Events are one of the most essential issues in an ADBMS, and thus their definition, their detection and their internal representation have received a big attention recently[CK+94, GJS92].

1.3 AMOS System

AMOS (Active Mediators Object System) is an *Object-Relational Active Database Management System.* It addresses support for future engineering information systems where autonomous, heterogeneous, and active databases and other software are distributed over fast computer networks. In such an environment active mediators simplify the communication between individual programs (usually being run on workstation) and the data sources from which information is retrieved. The purpose of these active mediators is to locate, transform, combine, query and monitor the desired information, and therefore retain flexibility and convenience for the user in very large federations of databases and other systems. This approach is called active mediators, since it includes active database facilities.
The AMOS architecture is built around a main memory based platform for intercommunicating information bases. Each AMOS server has full DBMS facilities, such as a local database, a data dictionary, a query processor, a transaction manager, and a communication manager.
A central component of AMOS is an object-relational query language, AMOSQL, with object oriented abstractions and declarative queries. The data model of AMOS and AMOSQL is strongly influenced by the functional data model OODAPLEX[Day89] and by the data model of Iris[FA+89] and OSQL[Lyn91]. The Iris data model has three basic constructs; objects, types and functions. The data model of AMOS extends that of Iris by introducing rules. Rules monitor changes to functions and changes to functions can trigger rules. Functions in AMOS can be stored, derived or foreign. Stored functions represent data stored as facts in the database (stored functions are internally represented as relations). Derived functions are AMOSQL queries (views) and defined in terms of other AMOSQL functions. Foreign functions are programs written in a foreign language (C or Lisp). Stored and derived functions in AMOS are updatable functions and changes to these functions are monitored. Changes to foreign functions are not monitored, they are supposed to be non updatable functions.(see section 2.1).
The AMOS kernel consists of several subsystems that are responsible for different tasks. The main subsystems are illustrated in Figure 2.1 and

include:

The external interface handles synchronous requests thought a client-server interface for loosely coupled applications and through a fast-path interface for tightly-coupled applications.

The command interpreter scans and parses AMOSQL expressions and sends request to the levels below.

The schema manager handles all schema operations such as creating or deleting types, i.e. object classes, and type instances including functions and rules.

The rule processor handles issues such as creation, deletion, activation, deactivation, triggering and execution of database rules[Skö94, SR96]. *The event manager* is integrated with the rule processor. It dispatches events to the rule processor. Events can come either from the external interface or from intercepted events in the lower levels such as schema updates or database updates.

*The foreign data source (DS) i*nterface of AMOS admits integration of foreign data structures and operators. Foreign operators are defined and implemented as multi-directional foreign functions with overloading on all arguments[LR92, FR96]

The query optimizer is responsible for transforming ad hoc queries, update statements, functions, and procedures into tractable execution plans using query optimization and compilation techniques[LR92, FR96, Flo96].

The execution plan interpreter handles the processing and execution of optimized expressions that are represented in the intermediate ObjectLog language[LR92].

The logical object manager manages all operations to objects in the database schema such as object creation, deletion and updates of object attributes including updating, inserting, and deleting data in stored functions. The level also manages OIDs (Object Identifiers) of the objects. An update operation causes the creation of an event that is intercepted and sent to the event manager.

The physical object manager includes parts for managing all physical operations on user objects (i.e. instances of user-specified types), system objects (strings, integers, reals, lists, arrays, vectors, atoms, hash tables, etc.) and event objects (objects representing database transactions). Examples of operations are allocation, deallocation, and access operations. Foreign functions can manipulate the physical object manager, e.g. to allocate and update user-defined internal storage structures.

The memory manager manages all the memory operations that automatically allocates and deallocates memory, and reclaims memory by garbage collection.

The disk manager in AMOS is more primitive in comparison to disk-based DBMSs since AMOS presupposes that the database resides in main-memory. It mainly handles flushing of database images between main-memory and disk for initiation, connection, or saving of databases.

The transaction manager controls all transactions the database by keeping a log of all database operations so that transactions can be undone or redone to guarantee database consistency.

The recovery manager is responsible for automatically maintaining persistency of a database that is exposed to transactions[Kar95].

The architecture of AMOS permits extensions to be made on the three levels of extensibility as identified in [CH90]: the *Data storage and access, the Query language,* and the *Query processing.*

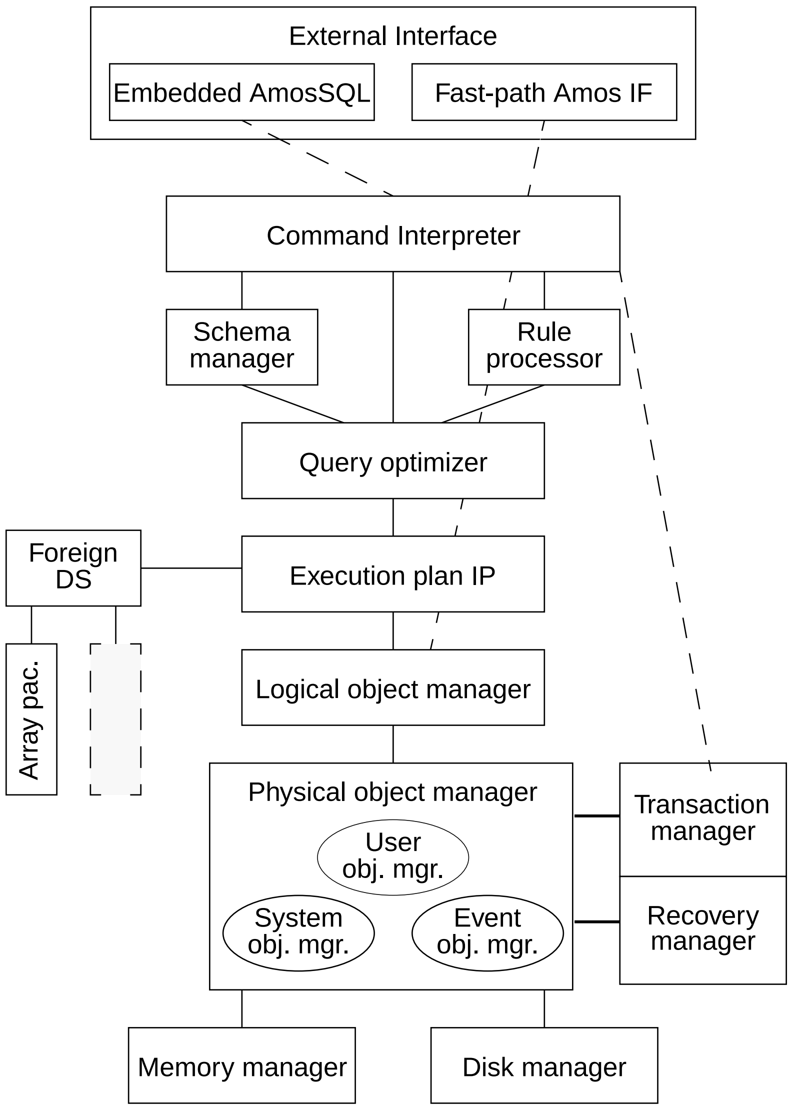


Figure 2.1: AMOS Architecture

1.4 Motivation and Summary of Contributions

The first version of the rule system of AMOS supported only the *Condition-Action* (CA) model by defining each rule as a pair <condition, action>, where the Condition is a declarative AMOSQL query, and the Action is any AMOSQL database procedure statement. The events involved are calculated from the condition. With the CA model the rule is triggered implicitly whenever data is updated so that new data satisfies the rule's condition.

Including a triggering event as part of the rule language of AMOS makes it possible to specify different actions when a given condition is satisfied, depending on which event occurred. For example, one might wish to react to violations of a referential integrity constraint in different ways, depending on whether the violation came about because a new object was added or because an old one was removed. This kind of operation-specific behaviour is not possible with Condition-Action rules. Significant work is done on integrating such a behaviour in the rule system of AMOS. A new syntax allowing specification of triggering events in the rule definition has been implemented and an Event Specification Language is defined. The events handled so far are: the creation of a new object, the deletion of an object, the insertion of a value into a bag-valued function, the deletion of a value from a bag-valued function, and the update of a function. An event is specified by its type and the involved object. Events can be simple or composite. Composite events are defined as event expressions, containing logical operators and events (simple or other composite events).

The integrated ECA rules are based on the concept of function monitoring; the event component as well as the condition component are compiled to AMOSQL functions. Only stored and derived functions can be referenced in the event specification. Foreign functions are assumed to be passive functions, i.e. functions that never change. The Action component is compiled to an AMOSQL procedure. Events are parameterized and data are passed from the event to the condition and from the condition to the action by using shared query variables.

The processing of a rule can be divided into four phases:

1. Event detection and Change maintenance
2. Rule triggering and Conflict resolution
3. Condition evaluation
4. Action execution

Event detection consists of detecting events that can affect any activated rules and is performed continuously during ongoing transactions. These events consist of changes to stored functions. A mechanism is built to record these changes and propagate them to the affected derived functions using Δ-relations and the incremental evaluation method. Rules are triggered whenever their specified events are detected and then inserted in a sorted queue for condition evaluation. Triggered rules are sorted based on priority numbers assigned to rule instances at activation time. Conditions are evaluated only for the updated data. During action execution further events might be generated causing all the phases to be repeated until no events are detected.

The event function is defined in terms of *delta relations*. These are generated by the rule compiler for each stored or derived function referenced in the event definition. Updates to stored functions are captured during the ongoing transaction by their corresponding delta relation and propagated to the affected derived functions incrementally through a dependency network at the *commit.* The propagation of changes is done through the network in a breadth-first, bottom-up manner and is based on incremental evaluation technique. Intermediate results are materialized to avoid the recomputation of derived functions during the rule processing since these might be referenced by more than one triggered rule.

Updates to tuples are handled directly instead of modelling them as deletions followed by insertions, this allows us to capture and difference between the three events addition, remove, and the update. The evaluation of the condition and the execution of the action are delayed till the end of the transaction, i.e. deferred coupling mode. The net effect of data modification operations during the transaction is considered and a calculus is defined to record only logical events in the delta relations.

1.5 Related Work

In the previous work dealing with integrating active rules in AMOS only CA-rules were considered. The involved Events were calculated from the condition by the rule compiler. The condition is an AMOSQL query and the Action can be any procedure statement, except commit. Data can be passed from the Condition to the Action of each rule by using shared variables.

Rules are furthermore parameterized and can be activated/deactivated for different argument patterns. The semantics associated with this syntax is as follows: If an event in the database changes the truth value for some instance of the condition to true, the rule is marked as triggered for that instance. If something happens later in the transaction which causes the Condition to become false again, the rule is no longer triggered.

The condition can specified over stored and derived functions only. The

events that trigger these conditions are the function update events, and adding or removing tuples to/from bag-valued functions.
The implemented rule processing of AMOS uses the database monitor method [Ris89] to detect changes on derived or stored attributes of database objects, however in [SR96] a more efficient technique for monitoring changes to rule conditions is proposed. This technique consists of generating several partially differentiated relations that detect changes to a derived function given changes to one of the functions it is derived from. Then to efficiently compute the changes of a rule condition based on changes of subconditions, the partially differentiated relations are computed by an incremental evaluation technique. A breadth-first, bottom-up propagation algorithm is also introduced to efficiently propagate both insertions and deletions without unnecessary materialization or computation.

The *Event-Condition-Action* (ECA) rule paradigm is widely accepted for active database systems, and it provides the flexibility required by most applications. All active database systems that support ECA rules provide, as basic features, *events* that correspond to data modification operations, *conditions* that correspond to queries over the database, and *actions* that are database operations.

HiPAC[CW96, HW93] is an active object-oriented database management system. Like AMOS it extends a basic object-oriented database management system with ECA rules based on the semantic model DAPLEX[Shi81] extended with the object-oriented features of OODAPLEX[Day89]. Rules in HiPAC, like all other forms of data, are treated as entities. There is a rule entity type, and every rule is an instance of this type. Special functions are defined over the rule type to fire, enable, or disable rules and like, other entities, rules can be created, modified or deleted.
In HiPAC events like rules are first-class entities; they are instances of the type event. The event type has two subtypes: primitive-event and composite-event. The primitive events are of three types: data-manipulation-events, time-events and external-events. HiPAC is an OODBMS, hence, all data manipulation occurs through the execution of functions on entities. To cause rules to be triggered when some data manipulation function is executed, a data-manipulation-event associated with the function has to be defined. Events can be defined for the generic data manipulation operations create, delete, and modify, as well as for type-specific operations; however, updates of derived functions (views) are not handled. Also, HiPAC allows operations for manipulating collections of entities. Composite events are defined by three parameterized types: disjunction, sequence, and closure.

Data manipulation events in HiPAC correspond to the execution of functions. Three basic techniques for detecting data manipulation events have been developed and incorporated into the HiPAC prototypes: the Hardwired, the Wrapper-based and the System-supported[CW96]. Concerning conditions which are pure queries and may refer to the parameters captured at the event occurrence, HiPAC uses three different techniques to suit the requirements of their evaluations: signal-driven evaluation, materialization of intermediate results, multiple condition optimization and incremental evaluation. The incremental evaluation is used in conjunction with the materialization of incremental results.

Starburst [CW96, HW93] is an extension of the Starburst relational DBMS at the IBM Almaden Research Center. The Starburst rule language is flexible and general, with a well defined semantics based on arbitrary database state transitions. Commands are provided for rule processing within transactions in addition to automatic rule processing at the end of each transaction. The event clause in the rule definition syntax specifies one or more events, any of which will trigger the rule. An Event is a relational data modification operation. The Condition is any SQL select statement and is true if the select statement produces one or more tuples. And the Action may be any database operation, including SQL data manipulation commands, data definition commands and the rollback.
The possible triggering events in Starburst correspond to the three standard relational data modification operations: inserted, deleted and updated. The updated triggering event may specify a list of columns, so that the rule is triggered only when of those columns is updated; specifying updates without a column list indicates that the rule is triggered by updates to any column.
Starburst supports transition tables which correspond to our Δ–relations. Views are specified as SQL select statements. Each view is computed once and stored as a database table (i.e. the view is materialized). A set of rules is generated automatically by the compiler from the materialized view definition. These rules are triggered by modifications to the base tables; their actions incrementally modify the materialized view according to the base table modifications.

The Ariel[CW96, HW93] rule language is a production rule language with enhancements for defining rules with conditions that can contain relational selections and joins, as well as specifications of events and transitions. Like in our rule language, only data modification type of events are considered in Ariel, however composite events are not handled.

A-RDL[CW96] uses Δ–relations to record the net effect of data modifications in a similar way to our approach. Updates to functions are handled directly and a calculus is defined to compute and record the net effect of changes in the delta relations.

CHAPTER 2

The AMOS Data Model

2.1 Data Model

The data model of AMOS and AMOSQL is strongly influenced by the functional data model OODAPLEX [Day89] and by the data model of Iris[FA+89] and OSQL[Lyn91]. The Iris data model has three basic modelling constructs; *objects*, *types* and *functions* and everything in the model is an object including types and functions. The AMOS data model extends that of Iris by introducing *rules*, a richer type system and multidatabase facilities[Fah94, FaR97]. Further more AMOSQL is influenced by the standardisation efforts such as SQL3[Mel95] and OQL[Cat94]. Rules are also objects and of type 'rule'. The relationship between objects, types, functions and rules in AMOS can be seen in figure 2.1.

Objects are used to model entities in the domain of interest. *Types* are used to classify objects and act as containers for their instances, i.e. *object classes*. All objects are instances of some type. Types themselves are of type 'type'. Objects can be created or deleted using the AMOSQL commands `create` or `delete` respectively.

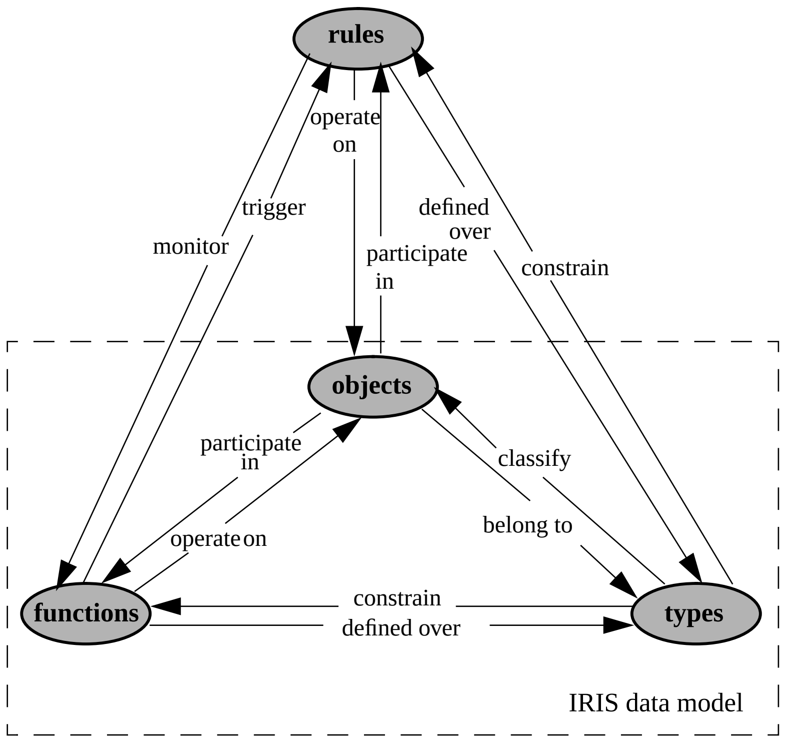


Figure 2.1: The AMOS data model

Functions are of type ‘function’ and are constrained to accept only objects that are instances of the declared argument type of any subtype thereof. They are used to model properties of objects and relationships between objects. Functions can be stored, derived or foreign. A stored function represents data stored as facts in the database. The corresponding mapping between arguments and results are internally stored in a table, i.e. relation. Stored functions can always be updated using the AMOSQL function update statements set, add, or remove[KF+95]. A derived function is defined by a single AMOSQL query (simple select statement). AMOS defines a derived function f to be updatable if it is derived from a single updatable function g in such a way that the argument and result parameters of f partition all the arguments and results of g and such that no selection is involved in the derivation[KF+95]. Foreign functions written in some procedural language(C or Lisp) are currently considered as passive functions which means functions that never change, such as built in arithmetic functions, boolean functions, and aggregate functions.

AMOS supports also *database procedures* which are defined by a program written in a procedural subset of AMOSQL that may have side effects, and overloaded functions which are functions defined on different types with identical names. Each specific implementation of an overloaded function is called a *resolvent*. When a function call is made to an overloaded function, the appropriate implementation, i.e. resolvent, is selected based on the actual argument types (*early binding*). Amos supports also *late binding* of overloaded functions where the overloaded resolution is done at run time instead of at compile time. (Examples of overloaded functions are given in section 2.3)

Rules are used to define constraints and are first-class objects; they inherit their operations from the *object* class. A rule can be created and deleted like other objects. **`create rule`** and **`delete rule`** commands are used for this purpose.

Two other operations are added to rules: **`activate rule`** and **`deactivate rule`** to enable and disable a rule respectively. Rules monitor changes to functions and changes to functions can trigger rules. All the events that rules can trigger on are modelled as changes to values of functions. Event functions, i.e functions that represent internal events, are defined over stored and derived functions, changes to these functions can affect the rule condition and trigger the rule.

Besides, *objects, types, functions* and *rules*, AMOS defines a set of other important types. Figure 2.2 below shows the AMOS type hierarchy:

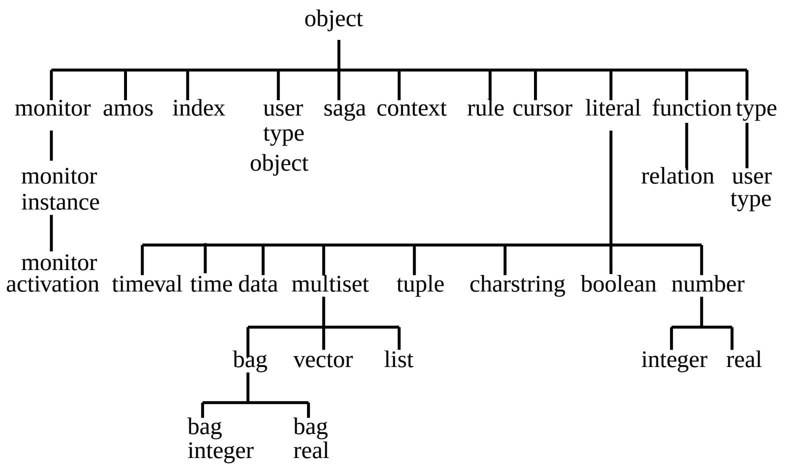


Figure 2.2: AMOS type hierarchy

Note that AMOS supports timestamps and defines three data types for referencing time. *Timeval* is a type for specifying absolute time points and *time* and *date* are types for relative time points.

2.2 Rule language

2.2.1 Rule definition

Integrating event specification in AMOSQL consists in defining a new syntax for rule definitions. This syntax should be conform to that of AMOSQL functions and should provide users facilities to specify the triggering events of the rule and also the possibility to define CA-rules by neglecting the event component of the definition.
In AMOS, rule processing is invoked automatically at the end of each user transaction (just before the *commit*) that triggers one or more rules. In additions, users can invoke rule processing within transactions by issuing the AMOSQL *Check* command. Hence, the minimum rule processing granularity in AMOS is a single database operation command and the maximum granularity is the entire transaction.

The implemented syntax for rule definitions is as follows:

```
create rule rule_name parameter_specification As
        [For_each_clause]
        [On event_specification]
        [When predicate_expression]
        Do procedure-expression
```

The **on** clause of the rule allows specification of the Event that will trigger the rule**.**
The **when** clause specifies the Condition that should be checked once the rule is triggered.
The **do** clause allows specification of the Action to execute when the rule is triggered and the condition is true.
In the For_each_clause, the rule's local variables are declared.

This rule is an event-based rule or what is commonly called *Event-Condition-Action rule* (ECA-rule). The meaning of such a rule is: "when an event occurs, check the condition and if it holds, execute the action".
If the event part is included, the condition part might be omitted, we then refer to an *Event-Action rule* (EA-rule). In this case, however, the condition is considered to be always true and the action is executed directly when the rule is triggered.
In some cases it may be useful to allow the compiler to generate the event definition. In this case, the event part is omitted and the user only specifies the condition and the action, and the system determines the events automatically. We then say we have a condition-based rule or a *Condition-Action rule* (CA-rule).
Remark: either the event part can be omitted or the condition part but not both in the definition of the rule.

The event_specification is the definition of the rule triggering event, it can be a simple event or a composite event. A composite event is a logical combination of simple events or other composite events. A simple event corresponds to a data modification operation and is specified by the event type and the modified object.(see chapter 4).

The predicate_expression is an AMOSQL query. It can contain any boolean expression, including conjunction, disjunction and negation. Furthermore, this query may refer to stored functions as well as to derived ones. If the query is non-empty then the condition is satisfied.

The procedure_expression in the rule action clause is any AMOSQL procedure statement, except *commit.*

2.2.2 Rule deletion

To delete a rule, we use the following syntax:

delete rule rule_name;

2.2.3 Rule activation and deactivation

Two operations are used to enable or disable a rule

activate rule rule_name parameter-list [**priority 0|1|2|3|4|5**]

and

deactivate rule rule_name parameter-list

respectively.

These commands allow certain rules to be “turned off” temporarily for the specified parameter list, so that the rules remain in the system but are not eligible to be triggered or executed.
Priorities are used for defining conflict resolution between rules that are triggered simultaneously; the default priority is 0, if the priority is not specified.

2.3 Example

The following example will be used later to illustrate rule definition and event specifications. We consider a simple database schema consisting of four objects types:

Address(street, postcode, city)
Department (name, addr, manager)
Employee (name, addr, dept, income, taxes, grossincome, netincome)
Manager (name, addr, dept., income, taxes, bonus, grossincome, netincome)

Employees are defined to have a name, an income, an address and a department. The netincome is defined based on taxes for both employees and managers, but with bonus for managers before taxes. Departments are defined to have a name, an address and a manager. The manager of an employee is derived by finding the manager of the department to which the employee is associated.

The AMOSQL schema is defined by:

```
create type address properties (street charstring,
                                postcode charstring,
                                city charstring);
create type department properties
        (name charstring, addr address);
create type employee properties
        (income number, taxes number);
create type manager subtype of employee;
create function name(employee) -> bag of charstring as stored;
create function addr(employee) -> bag of address as stored;
create function dept(employee) -> bag of department as stored;
create function bonus(manager) -> integer as stored;
create function grossincome(employee e) -> number as
        select income(e);
create function grossincome(manager m) -> number as
        select income(m) + bonus(m);
create function netincome(employee e) -> number as
        select employee.grossincome -> number(e) * taxes(e);
create function netincome(manager m) -> number as
        select grossincome(m) * taxes(m);
create function mgr (department) -> manager;
create function mgr(employee e) -> manager as
        select mgr(dept(e));
```

Note: The examples are somewhat unrealistic, but they serve to illustrate important aspects of rule definition and execution.

The functions grossincome, netincome, mgr, addr are overloaded on the types employee, manager and department, employee. For function calls grossincome(m), mgr(dept(e)) this is resolved at compile time, i.e. *early binding;* by the system using the local variable declarations. In some cases, the system cannot deduce what function to choose then the a ‘dot notation’, e.g. employee.grossincome->number(e), which specifies the types of the arguments and of the results can be used to aid the compiler to

choose the correct function at compile time.
In cases when the compiler cannot deduce the resolvent, it will produce a query plan that does run-time type checking to choose the correct function i.e. *late binding*. This would be the case if netincome was not overloaded and grossincome was specified without the 'dot notation'.
Basically functions can be single valued or *multi-valued* (bag valued). The later is indicated by the keyword 'bag of' in the result declaration. In our example, the functions name, dept and addr are multi-valued functions and hence, an employee can have more than one name, more than one department and more than one address. But at most he can have only one income and one taxes value since the corresponding functions income and taxes respectively, are defined as single-valued functions (by default, if the 'bag of' keyword is not indicated in the result declaration the function can take only a single value result).
Values can be added or removed to/from a bag-valued result updatable function by using the update statements add and remove respectively.
The set statement is used to update the value of an updatable function. The result of updating a function (even if it is a bag result function) by the command set is always a single value result. Internally, the set command is modelled as remove followed by an add. Applying a set on a bag result function, will first remove all the old values in the bag and then add the new value.

CHAPTER 3

Event Specification Language

Events are one of the essential issues in an ADBMS, and an ADBMS has to provide means for defining event types[DGA96]. Thus the definition, the detection and the management of events have received attention by many researchers and some Event Specification Languages have been proposed[GD93, CK+94, GJS92, CD91]. Events can be of different types: Data modification events, data retrieval events, time events and user-defined events. Only data modification events are considered in the current implementation of rules in AMOS. The creation and the deletion of an object, the insertion of a value into a multi-valued (bag result) function, the deletion of a value from a multi-valued function, the update of a function as well as the execution of rule actions, generate data modification events. However, temporal events which is an important class of events will be considered in the future extension of the rule language; the time points at which events are signalled are captured and recorded as a part of the event occurrence to facilitate this extension. Hence an event in AMOS[1] can be defined as “A database transition at a given time”. Events that are generated by commands issued either by interactive transactions or by transactions that are part of application programs connecting the database system are seen as external events. Events that are generated by rule execution are considered as internal events.

1. In AMOS, transaction time is used by timestamps

3.1 Simple events

The event can be any data modification operation caused by an AMOSQL command. The following types of simple events can be specified after the **on** clause:

. **Updated** (function_name(variable_name))
. **Added** (function_name(variable_name))
. **Removed** (function_name(variable_name))
. **Created** (variable_name)
. **Deleted** (variable_name)

where the type of the variable name is declared in the for_each_clause or in the arguments list of the rule definition (see section 2.2).

The *updated* triggering event is signalled whenever the specified function is altered by the AMOSQL function update command set, which sets the function to a new value[KF+95].

The *added* triggering event signalled whenever a tuple is added to the result of an updatable bag result function by the AMOSQL command add[KF+95].

The *removed* triggering event corresponds to the deletion of a tuple from the result of an updatable bag result function by the command remove [KF+95].

The *created* triggering event corresponds to any creation of a new instance of an object class using the AMOSQL command create[KF+95]. This includes the creation of a user-defined object, a function, a rule a type or any other AMOSQL object (see section 2.1).

The *deleted* triggering event corresponds to the deletion of an object and is raised whenever the AMOSQL delete command is executed. A syntax is defined to specify this event and the event manager detects its occurrence. However, the rule processor does not support references to deleted objects so far, due to the internal implementation of the delete command in AMOS. A special treatment is needed by the system when objects are deleted so that the event can be raised and the action executed before the object is physically deleted. We are working on this.

The *updated*, the *added* and the *removed* triggering events are represented on functions which can be either stored or derived.

● An example of a rule where the updated triggering event is specified on a stored function is:

```
Create rule rule1 (department d) as
    For each employee e
    On updated (income(e))
    When dept(e) = d and
            employee.netincome->number(e) > netincome(mgr(e))
    Do /*Action*/;
```

Example 3.1: updated event specification on a stored function

where the specified function income in the event clause is a stored function. The rule is triggered whenever the income of an employee is updated.

● An example of a rule where the event is specified on a derived function is:

```
Create rule rule2(department d) as
    For each employee e
    On updated(netincome(e))
    When dept(e) = d and
            employee.netincome->number(e) > netincome(mgr(e))
    Do /*Action*/;
```

Example 3.2: updated event specification on derived function

Here the updated event is specified on the derived function `netincome` which is a derived from the stored functions income and taxes (section 3.3).

● The *added* and *removed* triggering events are specified on updatable bag result functions. Here is an illustrating example:

```
Create rule rule3(department d) as
    For each employee e
    On added(addr(e))
    When dept(e) = d
    Do /*Action*/;
```

Example 3.3: Specifying the *added* triggering event

The specified function `addr` is an updatable bag result function. An

employee can have more than one address. The rule is triggered whenever an address is added to the bag of addresses of a given employee.
If the *removed* triggering event is specified instead of the added event in rule3, the rule will be triggered whenever an address is removed from the employee's bag of addresses.

● The *created* triggering event is specified on object instances of a given type. Let's look at this rule definition:

```
Create rule rule4(department d) as
    For each employee e
    On created(e)
    when dept(e) = d
    Do /*Action*/;
```

Example 3.4: *Created* triggering event specified on an object type

The rule rule4 is triggered whenever an object of type employee is created and the condition is evaluated to true if the new employee is set to belong to the department passed to rule4 during rule activation.

● Notice that rules in AMOS can be parameterized as it is illustrated in the previous examples. In some cases we may want to trigger our rule on updates of specific object instances; this can be done by passing the object in the argument list when the rule is activated. If several objects should be monitored the rule must be activated for each object.
Here is an illustrating example:

```
Create rule rule5 (employee e)
    On updated(income(e))
    When employee.netincome->number(e) > netincome(mgr(e))
    Do /*Action*/;
```

Example 3.5: event specification on a function for a specific object

The for_each_clause is omitted in this rule since the free variable in the event clause is declared in the argument list of the rule.
If we want to trigger this rule on updates on the income of an employee named :employee1, we activate the rule by this statement:

```
 activate rule (:employee1);
```

The event's specification variable "e" will be bound to the value :employee1 and only updates on the income of employee :employee1 triggers the rule (if it is not activated for other employees as well).

3.2 Composite events

Composite events are allowed by combining single (primitive) events or other composite events. AMOSQL is extended with an expression language for denoting these composite events using the logical operators OR and AND. Let's consider two events E1 and E2.

● The disjunction E1 OR E2 is an event that is signalled whenever E1 is signalled or E2 is signalled, the parameters of E1 OR E2 are the union of the parameters of E1 and the parameters of E2. `Rule6` below illustrates this:

```
Create rule rule6(department d) as
    For each employee e
    On updated(income(e)) OR updated(taxes(e))
    When dept(e) = d and
            employee.netincome->number(e) > netincome(mgr(e))
    Do /*Action*/;
```

Example 3.6: A composite event with the OR operator

This rule is triggered if the income of an employee has changed or the taxes are changed or both the income and the taxes are changed.

● The conjunction E1 AND E2 is an event that is signalled whenever E1 is signalled and E2 is signalled, no matter which one was the first (time sequence of events is not considered), the parameters of E1 AND E2 event are bound to the concatenation of the parameters of E1 and E2. For example, the rule `rule7` in example 3.7 is triggered whenever a change is detected on the address of an employee and on his taxes during the transaction.

```
Create rule rule7 (department d) as
    For each employee e
    On updated(addr(e)) AND updated(dept(e))
    When dept(e) = d and
            city(employee.addr->address(e)) !=[1] city(addr(d))
    Do /*Action*/;
```

Example 3.7: A composite event with the AND operator

● More complex composite events might be specified, in which case the left and the right parenthesis are used, i.e. ‘(‘ and ‘)’. For example:

```
On (updated(income(e)) OR updated(taxes(e)))
    AND (added(dept(e)) OR added(addr(e)))
```

Note that the parentheses have the priority over the `AND` operator which in turn has the priority over the `OR` operator, if we specify the composite event above without the parentheses for instance, we will get:

```
On updated(income(e)) OR updated(taxes(e)) AND added(dept(e)) OR
added(addr(e))
```

This event has a completely different semantics as the original event and it is interpreted by the system as:

```
On updated(income(e))
    OR (updated(taxes(e)) AND added(dept(e)))
    OR added(addr(e))
```

3.3 Condition Specification

The rule condition specified in the when clause is an AMOSQL query. It can contain any boolean expression, including conjunction, disjunction and negation. Further more, this query may refer to stored functions as well as to derived ones.
If the query is non-empty then the condition is satisfied. (see examples in sections 3.2 and 3.4)

1. In AMOSQL ‘!=’ means not equal

3.4 Action specification

The procedure_expression in the rule action clause can be any AMOSQL procedure statement, except *commit.* Examples of rule actions are:

```
Create rule rule8 (department d) as
    For each employee e
    On updated(income(e))
    when dept(e) = d and
            employee.netincome->number(e) > netincome(mgr(e))
    Do set income(e) = income(mgr(e));
```

Example 3.9: Action specification as AMSOQL statement.

In rule rule8, the action is specified as a simple AMOSQL statement that sets the income of an employee to that of his manager.

```
Create rule rule9 (department d) as
    For each employee e
    On updated(income(e))
    When dept(e) = d and
            employee.netincome->number(e) > netincome(mgr(e))
    Do SetIncome(e, income(mgr(e)));
```

Example 3.9: Action specification as a foreign function.

In rule9 *SetIncome(employee* e, number inc) may be a database procedure that sets the income of the employee e to the specified value inc.
The Action is executed only over the employees whose incomes have changed and caused the violation of the condition, i.e. set-oriented execution is supported[WF90], the action is executed on the set of tuples for which the condition is true.

CHAPTER 4 *Event Monitoring*

4.1 Delta relations

In AMOS stored functions are represented as tables(relations). In [SR96], Δ–sets are introduced as containers of logical changes to updated relations and are modelled as a pair of positive changes, i.e. set of added tuples to the relations, and negative changes, i.e. set of removed tuples from the relations. Updates to tuples are modelled as deletions followed by insertions. In our case, Δ-sets are redefined and used to materialize derived relations (see section 1.2) and record and maintain changes to functions. Updates to tuples are handled directly instead of modeling them to deletions and insertions.
We define the Δ-*set* of a stored or a derived relation R by:

ΔR = <R_*added*, R_*removed*, R_*updated*>

where, for some given transaction R_*added* contains at a point in time all tuples that are added to R as a result of the net effect of the transaction up to that point. Similarly R_*removed* contains all tuples that are deleted from R, and R_*updated* contains all tuples of R that are updated (both new values and old values are recorded).

4.2 Stored functions

The update operation of stored functions in AMOS is internally modelled as a delete of the old tuples followed by the addition of the new tuple. This is handled as one operation in the current implementation. Both the old value of the updated function and its new value are recorded in the corresponding delta-set. Doing this, we can reference the new value as well as the old value of the updated function in the rule Condition or in the Action and also be able to reset the function to its old value if the new one violates the defined constraint in the Condition part.

Let's take an example to illustrate how changes to stored functions are monitored using the redefined Δ-set. We consider changes to the stored function *dept* in the database sample in section 2.4. The *dept* function is defined as a bag result function so we assume that an employee can be in more than one department and initially the department of an employee :e1 is :toys. where :e1 and :toys are two AMOSQL variables of types employee and department, respectively, defined by the statements:

```
create department(name) instances :toys("toys_department");
create employee(name, dept)  instances :e1("employee1", :toys);
```

We suppose :shoes and :cloths are two other object instances of type department defined in the same manner as above.
When we do the update:

```
set dept(:e1) = :shoes;
```

at a given time t1 during the ongoing transaction, two operations are generated internally, first the tuple <:e1, :toys> is removed from the relation dept and second the tuple <:e1, :shoes> is added. However the event manager maps the two operations to one data modification operation or event and records both the old and the new value of the function dept by asserting the tuple <t1, :e1, :shoes, :toys> into the delta relation dept_updated.
If we update the department of :e1 by the AMOSQL statements:

```
remove dept(:e1) = :toys; /* at time t1*/
add dept(:e1) =  :shoes; /*at time t2 where t1 < t2*/
```

this yields the same result as before i.e. :e1 is in :shoes department in both cases; however this modification is modelled as two different events. The

first event is captured by the delta relation dept_removed by asserting the tuple <t1, :e1, :toys> to it and the second event is captured by the delta relation dept_added by asserting the tuple <t2, :e1, :shoes>.
Changes to a base relation R during the transaction are immediately captured and recorded by the Δ–relations R*_added,* R*_removed* or R*_updated* depending if the change is an insert, a delete or an update of a tuple in the relation R respectively. However, the net effect of changes is immediately considered. Table 4.1 takes the example of the department of an employee :e1 and shows how the net effects of the data modification operations on the relation dept are recorded in the delta relations during the ongoing transaction.
The time value at which the event happens is recorded in the delta relations as well, as explained in the previous chapter this helps for future extensions of the event specification language to include temporal events.

Table 1: Database modification operations and their effects on delta relations

Database modification operations	content of the relation[a]	Effects on delta relations
At t1[b]: <:e1, :toys> is added to dept At t2: <:e1, :toys> is removed from dept	<:toys> <>[c]	<t1, :e1, :toys> is in dept_added <t1, :e1, :toys> is not in dept_added
At t1: <:e1, :toys> is added to dept At t2: <:e1, :shoes> is added to dept At t3: dept(:e1) is updated to :cloths	<:toys> <:toys, :shoes> <:cloths>	<t1, :e1, :toys> is in dept_added <t2, :e1, :shoes> is in dept_added <t1, :e1, :toys> is not in dept_added <t2, :e1, :shoes> is not in dept_added <t3, :e1, :cloths, :shoes> is in dept_updated

Table 1: Database modification operations and their effects on delta relations

Database modification operations	content of the relation[a]	Effects on delta relations
At t1: dept(:e1) is updated to :shoes At t2: <:e1, :shoes> is removed from dept	<:shoes> <>	<t1, :e1, :shoes, :toys> is in dept_updated <t1, :e1, :shoes, :toys> is not in dept_updated <t2, :e1, :shoes> is in dept_removed>
At t1: dept(:e1) is updated to :cloths At t2: dept(:e1) is updated to :shoes	<:cloths> <:shoes>	<t1, :e1, :cloths, :toys> is in dept_updated <t1, :e1, :cloths, :toys> is not in dept_updated <t1, :e1, :shoes, :toys > is in dept_updated

a. content of dept(:e1) in our example
b. t1 represents the time at which the operation is issued. So are t2 and t3 where t1<t2<t3
c. empty bag

Formally if we consider a relation R, a tuple T, and we represent the delta relation R_added by Δ+R, the removed delta relation R_removed by Δ-R and the updated delta relation R_updated by Δ-+R then:

1. If T is added to R at time t1 and removed from R at time t2 where t1 < t2, the net effect is no modification: T is asserted into Δ+R at t1 but then retracted at t2.

2. If T is removed from R at t1 and added at t2, the net effect is no modification: T is first asserted to Δ-R at t1 but then retracted after the adding operation at t2.

3. If T is updated at t1 then removed from R at t2 where t1 > t2, the net effect is the removing operation. T is first asserted in Δ-+R at t1 but then retracted from it and asserted in Δ-R at t2 as a result of the removing operation.

4. If T is added to R at t1 and updated at t2 where t1 < t2, the net effect is the updated tuple. T is first asserted in Δ+R at t1 but then retracted and

asserted in Δ-+R at t2.

5. If T is updated at t1 and updated a second time at t2 then the net effect is the last updated T. The updated value of T, Tnew1, is asserted into Δ-/+R at t1 but at t2 Tnew1 is retracted from Δ–/+ and the new value of T, Tnew2, is asserted instead.

This is summarized in table 2 below:

Table 2: the net effect of database modification operations

t1	t2	net effect
-	+	Ø
+	-	Ø
+	-/+	-/+
-/+	-	-
-/+	-/+	-/+

4.3 Derived functions

Derived functions (views) are not updatable functions. These functions are defined in terms of stored functions. Events in our system as seen in the previous chapter, can be specified as updates of base (stored) functions as well as of derived functions. An event is raised whenever a tuple is added, removed or updated in the specified function. A derived function is typically recomputed every time it is referenced. This, however, can be very costly, since a derived function can span over large portions of the database.
The materialisation of derived functions is a technique to increase the performances of monitoring derived functions with respect to processor time[GM95]. Since in our system, derived functions as well as base functions may be referenced by different rule events at rule processing, then it may be beneficial to materialize and cache changes to these functions.
A materialized view is just as a cache, it gets dirty whenever the underlying base functions are modified. Updating a materialized view by recomputing

it from scratch is in most cases wasteful. Computing only the changes in a view to incrementally modify its materialization is much cheaper.
Incremental evaluation[SR96, HD91, GD93] is a technique that computes changes to derived functions by considering changes to base functions instead of computing them in full. This technique is used in our system in conjunction with the materialisation of incremental results to monitor changes to rule events. When an event is specified as an update of a derived function, the compiler generates Delta relations to the specified derived function as well as to all the underlying functions. A *dependency network* is derived from one or more rule events. This network takes as input, database modifications on stored functions and propagates the changes to derived functions that are affected. Figure 4.1 shows a dependency network for the derived function employee.netincome->number.

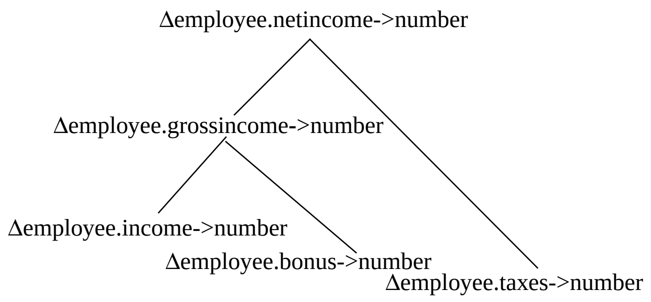


Figure 4.1: dependency Network

In the network of figure 4.1, all the dependencies of the function netincome are modelled as subnodes, changes to any of these base functions will affect the upper nodes. A propagation algorithms is used to propagate changes through the network from the bottom to the affected nodes in the upper levels using breadth-first, bottom-up propagation. This algorithm is based on incremental evaluation techniques (see section 5.3).

4.4 Event Functions

Events are detected based on the data changes stored in a temporary memory during the ongoing transaction. As seen above, Δ-relations are maintained for all stored or derived relations that are referenced in the event definition. The system generates at compile time *Event functions* for each specified rule event. These functions are defined in terms of the delta relations of the specified functions. Figure 4.2 below completes the dependency network in figure 4.1, by adding the event function node. During the ongoing transaction, updates to stored functions are recorded in their corresponding Δ-relations. At the check phase, these changes are propagated through the network to the derived functions and are recorded in their corresponding Δ-relations. The values of Event functions are then derived from the referenced Δ-relations.

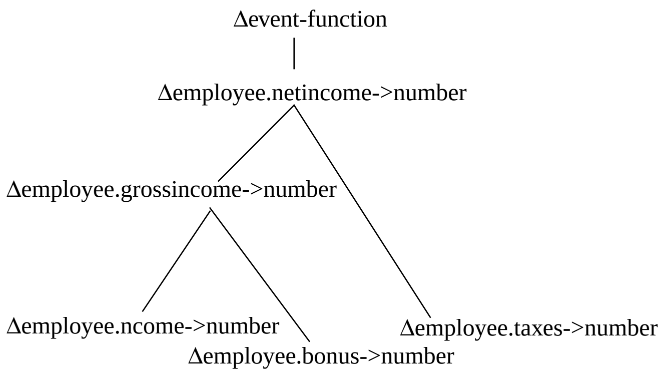


Figure 4.2: dependency Network of an Event function

In the following, we will see how the Event functions are derived from the Event definition and how they are defined in terms of the Δ-relations of the specified functions.

4.4.1 Simple events

● Let’s consider the following rule:

```
Create rule no_high(department d) as
      For each employee e
      On updated (income(e))
      When dept(e) = d and
                employee.netincome->number(e) > netincome(mgr(e))
      Do set employee.grossincome->number(e) = grossincome(mgr(e));
```

The Event as defined in rule no_high is compiled into a function (example 4.1) represented as an AMOSQL function, *evt_no_high*, that returns only the employees whose incomes are updated since it uses in its definition the Δ-relation *income_updated* generated for the income function at compile time and which contains only the updated tuples.

```
create function evt_no_high()->employee e as
      select e for each number n, timeval t
      where timeval.employee.income_updated->number(e)@t = n;
```

Example 4.1: The generated event function for a simple updated event.

Notice the specification of the parameter declaration timeval in the *evt_no_high* function. This is defined in AMOS as a *timestamp* and represents in the function the transaction time at which the income of a given employee is updated. The timeval as explained in chapter 4 is recorded in the delta relation income_updated with the parameters of the raised event. Notice also that the delta relation income_updated is defined as an AMOSQL function with two argument types timeval and employee and the result type number. The notation ‘@’ means ‘at’ and it is just another way of specifying parameters in a function call when we deal with the type timeval.
timeval.employee.income_updated->number(e)@t = n in the where clause can be replaced by the common function call syntax for all AMOSQL types timeval.employee.income_updated->number(t, e) = n.
The condition is compiled into a condition function and it looks like:

```
create function cnd_no_high(department d, employee e)-> employee
         as select e where dept(e)= d and
         employee.netincome->number(e) >
                                              neticome(mgr(e));
```

The function `cnd_no_high` is evaluated with all the parameters to the rule instantiated, in this case with the department d instantiated, and all the returned changed data from the event function `evt_no_high`, therefore the condition is evaluated only on those data that have been updated and caused the rule to trigger.

The AMOSQL action procedure generated for the action in the rule `no_high` looks like:

```
create function act_no_high(employee e)-> boolean as
          set employee.grossincome->number(e) =
                                        grossincome(mgr(e));
```

The function `act_no_high` is executed over the returned parameters from the condition function, i.e. the subset of the data which verify the condition predicates. Semantically it can be seen as:

```
for each department d
where d = no_high_activations()
   call act_no_high(cnd_no_high(d, evt_no_high()));
```

where no_high_activation is a function that returns all the arguments for which the no_high rule is activated, i.e. set-oriented rule execution[CW96].

The general expressions of the generated functions for the different clauses of the rule, the event, the condition and the action clause can be represented as:

```
          evt_rule_name(|<rule_arg>|) -> evt_res
          cnd_rule_name(|<rule_arg>,| <evt_res>) -> cnd_res
          act_rule_name(|<rule_arg>,| <cnd_res>)
```

● Let’s look at some different illustrating examples which show the generated monitoring functions for each type of events and the corresponding condition functions:

```
Create rule rule1() as
   For each employee e
   On updated(mgr(e))
   When netincome(mgr(e)) < 20000
   Do /*Action*/;
```

The Event in rule1 is specified on the derived function mgr having the signature employee.mgr->manager and which is derived from the base function department.mgr->manager (see section 2.3).

The generated Event function for this rule will look like:

```
Create function evt_rule1() -> employee e as
        select e for each manager m, timeval t
        where timeval.employee.mgr_updated->manager(e)@t =m;
```

and the condition function as:

```
create function cnd_rule1(employee e) -> employee as select e where
        employee.netincome->number(e) > netincome(mgr(e));
```

● Create rule rule2(department d, employee e)
    On updated *(income(e))*
    When dept(e) = d
    Do /*Action*/;

In rule2, the type of the free variable 'e' in the Event definition is specified in the argument list of the rule. The generated Event function for this rule is:

```
create function evt_rule2(employee e) -> employee as
        select e for each number n, timeval t
        where timeval.employee.income_updated->number(e)@t = n;
```

This event function takes the specified parameter as an argument and returns it if it is found in the specified delta relation.
The condition function looks like:

```
create function cnd_rule2(department d, employee e) -> employee as
        select e where dept(e) = d;
```

● The *added* and *removed* triggering events are specified on bag result functions. Let's consider again the rule *rule3* in example 3.3 where the *added* triggering event is specified on the function addr of an employee. The generated Event function will be as is shown below:

```
create function evt_rule3 (employee e) -> employee as
        select e for each timeval t, address a
        where timeval.employee.addr_added->address(e)@t=a;
```

and the condition function is:

```
create function cnd_rule3(department d, employee e) -> employee as
        select e where dept(e) = d;
```

The event function `evt_rule3` returns all the employees whose addresses have been updated by adding a new address.

● A creation of an object is seen as an insertion to the built-in AMOSQL function `Allobjects` that returns all the objects in the system. The create object event is mapped to an added event to the system function `Allob-jects.` The object creation is in the kernel of the system and therefore `Allobjects` is a foreign function.
Let’s consider the rule `rule4` in example 4.4 again. The triggering event for this rule is defined as:
On created(e), where e is any object of type employee.

Creating an object instance of type employee is mapped to an insertion of an object of type employee to the relation `Allobjects`. Hence, the generated event function will look like:

```
Create function evt_rule4() -> employee e as
        select e for each timeval t
        where timeval.allobjects_added->object()@t=e;
```

A Δ-relation is generated to the function `Allobjects` when the `Cre-ated` triggering event is specified in any activated rule. This Δ-relation maintains all the created objects of any types during the ongoing transaction and is cleared after the *commit* or the *check* as the other Δ-relations.

The condition function is:

```
Create function cnd_rule4(department d, employee e) -> employee as
        select e where dep(e) = d;
```

The event function evt_rule4 is specified in terms of the generated delta relation by the rule compiler allobjects_added. It returns all the objects of type employee that have been created during the current transaction (since the last *commit* or *check*). At the monitoring check time we can detect if a new employee has been added to the extent of the type employee and pass it as a parameter to the condition function. The Condition function evaluates the condition predicates for the department instantiated in the rule argument list and returns the added employee to the Action function.

4.4.2 Composite events

Composite events are logical combinations of simple or other composite events. The generated functions for this class of events may contain in their definitions conjunctions and/or disjunctions of the referenced Δ-relations.

● Let’s consider again, rule6 in Example 3.6. The event in this rule is defined as:

On updated(income(e)) ***or*** updated(taxes(e))

This event is defined as a disjunction of two single events: the updated triggering event on the stored function income and the updated event on the stored function add. Hence, the generated event function for this event will contain in its definition a disjunction between the Δ-relation `income_updated` and the Δ-relation `addr_updated.` This is shown below:

```
Create function evt_rule6() -> employee e as
     select e for each timeval t1, timevalt2, number n1, number n2
    where employee.income_updated->number(e)@t1= n1
    or timeval.employee.taxes_updated->number(e)@t2=n2;
```

The condition function will look like:

```
Create function cnd_rule6(department d, employee e) -> employee as
   select e
   where dept(e) = d
   and employee.netincome->number(e) > income(mgr(e));
```

The Event function returns all the employees whose incomes or taxes have been changed.

● In the rule rule7 in example 4.7. The event is defined as a conjunction of simple events as below:

On updated(addr(e)) ***and*** updated(taxes(e))

For this rule the Event function will contain in its definition the logical operator ‘and’ between the updated Δ–relations generated for the referenced functions `taxes` and `addr.` The returned parameters from this function will be the intersection of the set of employees whose taxes have changed and the set of the employees whose addresses have been changed as well during the transaction.
The generated event function is shown below:

```
Create function evt_rul7() -> employee e as
        select e for each timeval t1, timevalt2, number n, address a
        where employee.taxes_updated->number(e)@t1= n
        and timeval.employee.addr_updated->address(e)@t2=a
```

4.4.3 CA-rules

Let’s consider the following rule:

```
Create rule ruleCA(department d) as
    For each employee e
    When dept(e) = d and
            employee.income->number(e) > 10000
    Do set income(e) = 10000;
```

At compile time an Event definition is generated for the rule from the Condition. This Event is specified as a disjunction of updates on all the referenced relations in the Condition. Then an event function is created which contain in its definition a disjunction of all the updated Δ-relations generated for the referenced functions in the event definition.
The condition references two functions `income` and `addr`, the triggering event can be specified as:
`on updated(income(e)) or updated(dept(e))`
Therefore, the generated Event function for the this rule will be:

```
create function evt_ruleCA(department d) -> employee e as
      select e for each employee e, timeval t1, timeval t2, number n
       where timeval.employee.dept(e)->department@t1 = d
      or employee.income_updated->number(e)@t2 = n;
```

The condition is compiled into a condition function represented as:

```
create function cnd_ruleCA(department d, employee e) -> employee
    as select e
    where dept(e) = d and
    employee.income->number(e) > 10000;
```

and the action into the action function act_ruleCA shown below:

```
create function act_ruleCA(employee e)->boolean e as
    set employee.income->number(e) = 10000;
```

With these three generated functions, the rule will be processed just as an ECA-rule, the event function is first executed. The returned parameters are passed to the Condition function. If the Condition function is evaluated to true after that then the action procedure is invoked.

CHAPTER 5

Implementation Issues

5.1 Rule Processing Algorithm

Rule processing in AMOS is invoked automatically at the end of each user transaction (just before the *commit*) that triggers one or more rules or within transactions by issuing the AMOSQL *Check* command. Hence, the minimum rule processing granularity in AMOS is a single database operation command and the maximum granularity is the entire transaction.
During rule processing, the first time a triggered rule is executed it considers all modifications made by rules. If the rule is triggered additional times, it considers all modifications since the last time it was checked (because the Δ–sets are cleared after each check phase).
The basic rule processing algorithm in AMOS is described as follows:

```
1. detect events
2. mark triggered rules (put the rules in a sorted queue)
3. pick the rule having the highest priority
4. evaluate the rule's condition
5. act (if the condition is true, execute the action)
6. repeat from 1 until no more rules in queue and no
        more new events are raised
```

Algorithm 1: Rule processing algorithm

Events are detected based on the data changes and are stored in a temporary memory (the Δ–sets) during the ongoing transaction. Event functions are generated for each specified event at rule compilation and Δ-sets are created for each referenced stored or derived function by the event function. During the ongoing transaction, updates of stored functions are recorded in these Δ-sets. At check time, changes on stored functions are propagated through a network to derived functions and then event functions are marked as changed if they are affected.
A subset of rules are triggered and put in a sorted queue based on numeric priorities. Rules are assigned ordering priorities from 0 to 5, hence, when a triggered rule is selected for condition evaluation and possible execution; it is selected such that no other triggered rule has a higher priority.

In addition to the automatic rule processing at the transaction commit (E-C deferred mode)) rule processing is invoked within transactions when the user issues the *Check* command. The Check command invokes the same rule processing algorithm that is invoked at transaction end. Regardless of whether a rule is executed in response to one of these commands or in response to end-of-transaction rule processing, the semantics is the same: The rule considers the entire set of the recorded modifications since it was last considered within the transaction, or since the start of the transaction if it has not yet been considered.
If a rollback is executed in a rule action, then rule processing terminates and the transaction is aborted (the added tuples to the Δ-sets are removed).

5.2 Creation of a rule

5.2.1 Delta relations

At rule creation, Δ-sets are created for the specified stored functions in the event part of the rule definition.
Let’s take an example of a simple event:

```
        On updated(income(e))
```

In this event the AMOSQL function `income` is specified, so at compile time, a Δ-set is generated for the function `income.` This Δ-set contains three components: `income_added` that holds all the added tuples to the relation `income`, `income_removed` contains all the removed tuples from the relation income, and `income_updated` which maintain the set of the updated tuples. This can be represented as:

Δincome = < income_added, income_removed, inocme_updated>

In case derived functions are specified, Δ-sets are generated for these derived functions and all the underlying stored or derived functions.
Let’s consider this event:

```
        On updated (netincome(e))
```

In this event the derived function `netincome` is specified. The `netincome` function uses in its definition the derived function `grossincome` and the stored function `taxes`. The derived function grossincome is defined on the stored function *income* (see section 2.3), then Δ-sets are created for all the involved functions, `income, taxes, grossincome` and `netincome,` and are inserted into a propagation network.

5.2.2 Event functions

The event part in the rule definition is compiled to a function that uses in its definition the generated Δ-relations for the involved functions in the specification of the event as explained above. These Δ-relations are containers of the changed data, they return all the modified data. Hence, the definition of the event function on Δ-relations instead of the complete functions ensures a correct and an efficient execution.
In the case of CA-rules, where only the condition and the action are specified by the user, the compiler creates Δ-relations for all the involved derived and stored functions in the definition of the condition and generate an event function in terms of these Δ-relations (see section 4.3).

5.2.3 Condition functions

Condition functions are generated as well to the condition part of the rule. These are ordinary AMOSQL functions. A condition function takes its parameters from the argument list of the rule and the returned parameters from the event function. This means that the condition function is executed only over the modified objects. The returned objects are a subset of the instantiated objects in the argument list that the condition references (shared with the Event part). In the case of an EA-rule all the passed parameters are returned since the condition doesn’t contain any predicate expression and returns all the passed parameters..

5.2.4 Action procedures

An action procedure is generated for the action part of the rule. The free variables in the Action are calculated at compile time and specified as argument in the action function. The values of these arguments might be passed from the rule activation arguments and in the most general case are instantiated to the returned values from the condition function (shared variables with the Condition part). In case the Action does not contain any free variable (rollback for instance), a dummy variable is passed from the condition to the action function to ensure the correctness of the rule execution and its semantics.

5.3 Propagation network

The propagation network contains information needed to propagate changes affecting activated rules. Since the propagation is done in a breadth-first, bottom-up manner the network can be modelled as a sequential list, starting with the lowest level and moving upward. Each level consist of the list of network nodes
Each Δ-relation affecting activated rules is associated with one (and only one) node consisting of (see figure 5.1 below):
-A change flag, chg-flg, marking the node as changed
-A counter, cnt, that states how many times the node is propagated during the check phase
-The Δ-set of the relation
-The relation
-A list of affects nodes, a-list, that are affected by changes to this node
-A list of depends on nodes, d-list,

chg-flg	cnt	Δ-set	relation	a-list	d-list

Figure 5.1: The network node data structure

The number of levels needed in a network depends on how relations are expanded.

Nodes associated to stored relations are inserted in the bottom level, those associated to the event functions of the activated rules are inserted in the top level and the derived functions nodes are inserted in the intermediate levels. For late binding extra levels are inserted in such a way that the stored relations nodes are always inserted in the bottom level and the nodes associated with event functions are stored in the top level. Figure 5.2 shows how nodes are connected in a 4 level network:

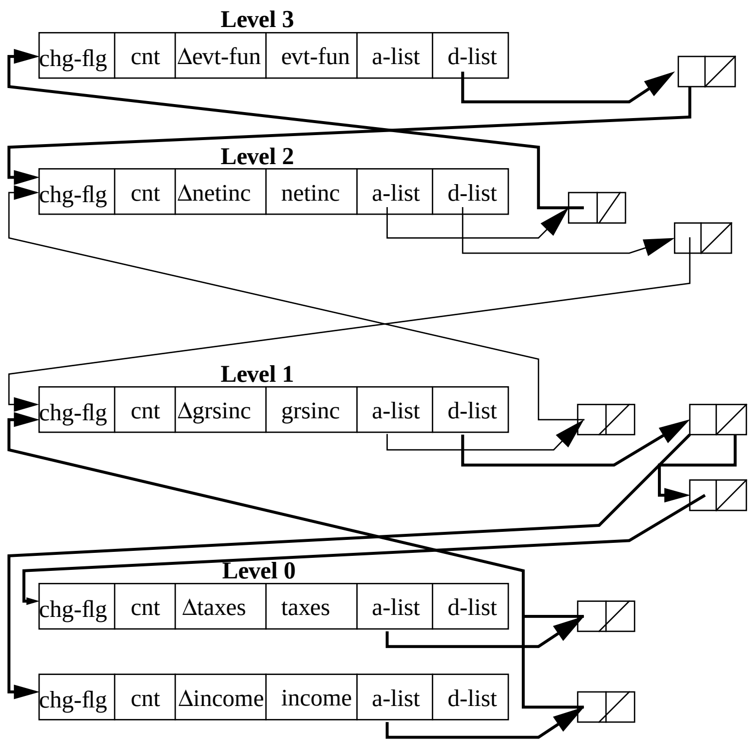
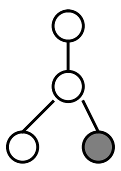
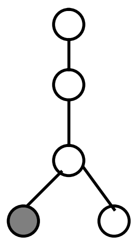
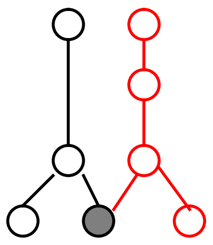
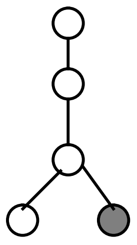
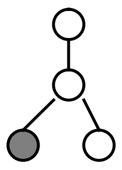
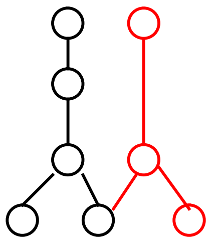

Level 3
chg-flg | cnt | Δevt-fun | evt-fun | a-list | d-list
Level 2
chg-flg | cnt | Δnetinc | netinc | a-list | d-list
Level 1
chg-flg | cnt | Δgrsinc | grsinc | a-list | d-list
Level 0
chg-flg | cnt | Δtaxes | taxes | a-list | d-list
chg-flg | cnt | Δincome | income | a-list | d-list

Figure 5.2: The nodes for the propagation network for the event ”on updated(netincome(e))”

5.4 Activation/Deactivation of a rule

At rule activation, the rule activation is inserted into a propagation network. This is done by inserting the dependency network of a rule's event function. Depending on the definition of specified functions in the rule's event, a rule might need more levels than the initial network contains, therefore, the topology structure of the network is modified; levels are added or removed (see table 3 below) to fit all the nodes and keep the dependency relationship between the structured functions. The nodes of the stored functions are always stored in the bottom level of the network and the nodes corresponding to the event functions are in the top of the network.

TABLE 3. Insertion of an event function node into a propagation network

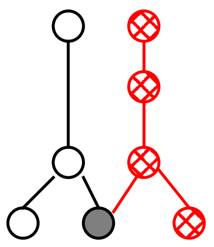
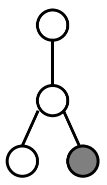
Before the insertion	**The dependency network to be inserted**	**After the insertion**
		
		

The algorithm for inserting Δ-relations into the network is as follow:

```
Insert(ΔP):
    If ΔP is not already inserted into the network then
    get node_of(ΔP);
    if Dp is empty, where Δp is the set of relations that
        P depends on,
    then /*P is a base relation*/
        Insert_in_level (node_of(ΔP), 0);
    else
        for each ΔQ where Q in Dp do
            Insert(ΔQ);
            Insert node_of(ΔQ) into the
                depends-on list node_of(ΔP).d-list;
            Insert node_of(ΔP) into the affects list
                node_of(ΔQ).a-list;
        Insert_in_level(node_of(ΔP),
            max(for each ΔQ where Q Dp:
            level_of(node_of(ΔQ))) +1);
```

At rule deactivation, the rule and its activation is removed from the network. The topology of the network is restructured too whenever a rule is removed from it. Table 4 below gives a simple example where a an event function node is removed from the network and one level in the resulted network is removed since it is no more needed.

TABLE 4. **Deletion of an event function node from the propagation network**

Before the deletion	**After the deletion**
	

The algorithm for removing Δ-relations from the network is:

```
Remove(ΔP):
    if ΔP is present in the network then
        if the affects list node_of(ΔP).a-list is empty then
            for each ΔQ where Q Δp
                remove (node_of(ΔQ) from
                    the depends-on list
                node_of(ΔP).d-list;
                remove (node_of(ΔP) from the
                    affects list node_of(ΔQ).a-list;
                if node_of(ΔQ).a-list is empty then
                    Remove(ΔQ);
            Remove_from_level (   node_of(ΔP),
                                  level_of(node_of(ΔP)));
```

5.5 Data modification

When a base relation is updated, the new tuple is recorded in the Δ-set with the time value at which it is held. If a tuple is added to a relation then the tuple is asserted in the Δ-added component of the relation Δ-set. If the tuple is removed from the relation then this tuple is asserted in the Δ-removed component of the Δ-set and if the tuple is updated then it is asserted in the Δ-updated part of the relation Δ-set. However the net-effect of changes during a transaction are considered right after the update operations and before these changes are recorded into the delta relations.

5.6 Check phase

The rule processing is done in three phase:
1. Changes are propagated through the network of activated rules from the base relations to the derived relations. The event function nodes affected by these changes are marked as changed.
2. All the activated rules whose nodes are marked as changed during the propagation are marked as triggered and are sorted in a triggered rules queue based on priority numbers.
3. The triggered rules are picked up from the queue and fired one after another. The condition is evaluated by executing the condition function. If a

non empty result is returned then the action is executed.
The execution of an action may trigger new rules or cause old triggered rules to be no more triggered, hence the processing algorithm is reexecuted after each rule execution and the queue of triggered rules is updated in each cycle.

The check phase algorithm looks as follows:

```
Check():
    propagate();
    while more rules in the queue or new triggered rules
        execute event function for each node marked as
            changed in the top of the network;
        insert the rule with its activation and the returned
            result from the event function in the sorted trig-
            gered rules queue;
        Trigger_rules();

Trigger_rules():
    get the rule with the highest priority in the queue;
    execute the rule Condition;
    if a non-empty result is returned execute the Action;
```

5.6.1 Propagation Algorithm

In the check phase the propagation algorithm propagates all the non-empty Δ-sets in a breadth first manner from bottom to the top, as illustrated in figure 5.3. Since the network is constructed in such a way that the change dependencies of one node, i.e. the Δ-relations it depends on; are calculated in the network levels below, a breadth-first propagation ensures that all the underlying nodes for a given node are checked if changed before this node is reached and hence any change to a node in the network is detected and propagated to all the affected nodes efficiently.
The propagation is done in two steps. In the first step all the affected nodes by changes on stored relations, i.e. nodes in the bottom of the network; are marked as changed by setting the chg-flg to true.
In the second step changes are propagated to all the nodes marked as changed in the first step from the bottom nodes to the upper nodes in a breadth-first manner. A compute algorithm is invoked for each reached node to maintain changes on the corresponding relations. However, this algorithm assumes that the derived functions have the same parameters as the func-

tions they depend on, which is rather restricted. Changes to other kinds of derived functions are not monitored so far. even though in the first step of our algorithm, changes to these functions can be detected and the corresponding events can be raise. As an example of such functions we consider an event that is defined on a derived function which returns all the employees in a given department whose netincomes are higher than the netincome of their manager. This function can be defined in AMOSQL as:

```
create function high_incomes (department d) -> employee as
     select e for each employee e
     where dept(e) = d and
             employee.netincome->number(e) > netincome(mgr(e));
```

The function high_incomes is a derived function and is defined in terms of the stored function employee.dept->department and the derived functions employee.netincome->number, manager.netincome->number and employee.mgr->manager. All these underlying functions take as argument parameters of types employee or manager (see section 2.3). However, high_incomes takes as argument a parameter of type department. Changes to any of the underlying functions of high_incomes can be detected, computed and recorded in the corresponding delta relations however, these changes can not be propagated to the function high_incomes itself; our algorithm as implemented currently can not find out the involved department. Different solutions might be figured out to handle this type of derived functions and an efficient algorithm that computes changes to any derived function given changes to one of the stored function it depends on is needed. In [Skö94, SR96], such an algorithm is introduced and used to propagate changes partially and incrementally from base relations to the rule conditions. A similar approach might be used to propagate changes to events functions defined on more complex derived functions in a future work.

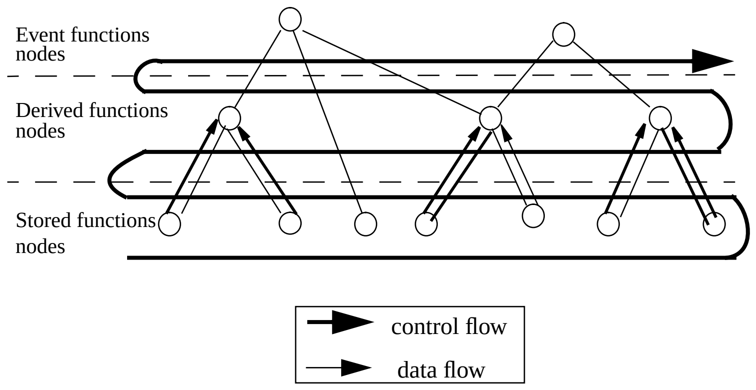


Figure 5.3: Propagation by breadth-first algorithm

The propagation algorithm looks as follow:

```
propagate():
    for each layer l in the network starting from level 0
    mark-level-changed(l);
    for each layer l in the network starting from level 1
    propagate-layer(l);

mark-level-changed (l):
    for each node n in the layer l
    mark-node-changed(n);

propagate-layer (l):
    for each node n in the layer l
    propagate-changes(n);

mark-node-changed(n):
    if chg-flg(n) = true then
        for each above-node in node.a-list do
        set chg-flg = true;
```

```
propagate-changes(n):
    if node chg-flg(n) = true then
        compute-deltas for n;

compute-deltas(n):
    for each delta-set in node.d-list of n
        execute node.relation with the delta-set parameters;
```

5.6.2 Rule triggering and conflict resolution

At propagation phase of the network, the reached rule event nodes are marked as changed. All the corresponding rules to these nodes are inserted in a triggered rules queue. The queue is modelled as a list of triggers. A trigger is defined as a list containing the triggered rule object, a bag of the event function parameters, its returned results, and the priority number of the rule. This can be represented as:

```
Trigger: (rule-object,
          <event-function-parameters, event-function-results>,
          priority number)
```

where the event-function-parameters list may contain the rule activation if the rule arguments are referenced in the definition of the event.

Rule instances are assigned priority numbers by the user at the activation time or the default priority number, 0, by the system. Before inserting a trigger in the queue, its position with respect to the already existing triggers in the queue is computed. This depends on its priority number.

5.6.3 Condition evaluation and rule execution

Once a rule is pulled from the queue, its condition function is executed for the instantiated rule argument list at activation and the returned parameters from the event function. If the condition function returns a non empty result then the action procedure is executed for the returned results from the condition function.

5.6.4 Termination

The rule processing algorithm as shown above is iterative. The loop continues until there are no more triggered rules. However, the execution of some rule actions can produce events that trigger other rules or trigger the same rule again and then it is possible for rules to trigger each other indefinitely. Several ways have been proposed to handle termination [Bou94, CW96]. In the current implementation of the rule processing algorithm an upper limit on how many times a rule can be executed during rule processing is established. If this limit is reached, rule processing terminates abnormally. This upper limit is established as a system parameter and can be set by the user at the check time. The default value is 20.

Note: all the algorithms presented in this chapter are implemented in AMOSQL-Lisp[FKR95], the internal Lisp interpreter of AMOS.

CHAPTER 6

Conclusions and Future Work

This report presents a significant extension of an early work done on integrating active rules in the Object relational Database Management System, AMOS. The extension consists in integrating Event Specifications in the rule language of AMOS. A new syntax for rule definition has been implemented and the definition, detection and management of events are investigated. The Event specification language considers only database modification events so far. These consist of the creation and the deletion of an object, the insertion of a value to a function, the deletion of a value from a bag-valued function and the modification of a function value. Events might be specified as simple events or composite events. The composite events are combinations of simple or other composite events. The conjunction form and the disjunction form are used to define composite events. The rule syntax allows to define both ECA and CA rules. ECA rules are triggered based on detecting and capturing the specified events during a transaction. CA rules are compiled to ECA rules after calculating the involved events from the Condition by the rule compiler and generating the triggering event expression.
Rules are based on the concept of function monitoring. All the changes of the system that the rules are to monitor are introduced as changes to functions. Events might be specified on both stored and derived functions and are compiled to active functions defined in terms of the relations they depend on. To efficiently monitor changes on the events functions, the rule compiler generates delta relations that capture changes to derived functions given

changes to one of the functions it is derived from. The changes are computed by incremental evaluation techniques using a propagation network.
The main contributions of this work can be summarized in the following:

1. Introducing ECA rules paradigm in a functional OO model, AMOSQL.
2. Defining an Event Specification Language for defining simple or composite events.
3. Introducing Event functions that represent the internal implementation of the specified events.
4. Mapping changes to stored functions to tables, i.e delta relations; and recording all data modification of these functions in their corresponding tables.
5. Monitoring changes to event functions defined in terms of derived functions uses an incremental evaluation technique, where delta relations are generated for derived functions and changes are propagated in a breadth-first manner to these relations whenever the underlying functions have changed through a propagation network.
6. Handling object creation as events on system functions and mapping these functions to delta relations that record the created objects.
7. Compiling rules of type CA to ECA rules by calculating the involved events and generating the monitoring Event functions.

There are a number of remaining issues, both practical and theoretical, that need to be addressed to provide a complete event specification language in AMOS:

1. The incremental evaluation technique presented here needs to be extended to include complex derived functions. The actual implementation supports only a specific class of derived functions where the derived functions are constrained to have the same argument list as the functions they depend on. Partial differentiation technique as described in [SR96] can be used to deal with more general and complex derived functions.
2. A more efficient way of handling rules of type CA is by monitoring changes to the rule condition rather than generating an event expression from the condition and processing the rule as an ECA rule. Significant work on Condition monitoring using partial differentiation is done in [SR96, Skö94], a similar approach can be integrated with our approach to handle CA rules.
3. Contexts or rule sets have been introduced in [SRF95] and integrated in the old rule system of AMOS. A similar mechanism is needed for organizing and structuring ECA rules into sets that can be activated and deactivated

dynamically. The system monitors only those events that affect rules of activated sets.
4. Only data modifications events are supported so far; temporal events can also be included. The extension of the current system to include temporal events is studied, the time of the event occurrence is captured and recorded in the delta relations. However a specification language for temporal events is needed and how the incremental change monitoring techniques relate to time events must also be investigated further.
5. Events are currently specified only on stored and derived functions. Foreign functions are assumed to be not updatable in AMOS (except for the system function allobjects). However, in some applications foreign data sources need to be monitored. Such data might originate from physical sensors, external pieces of software or as in the case of the stock exchange, a transaction system with its own database. Foreign data sources can be presented in the database as if they are local data and the database should support access, monitoring and updates in a transparent manner. Such data sources might be represented in AMOS as foreign functions and a further investigation is addressed to see how delta relations can be used efficiently to capture changes on these functions and trigger rules.
6. Composite events can be extended to include the negation operator and operators such as ‘before’ and ‘after’ for instance to specify sequenced conjunctive events mainly when time events are considered.
7. The rules in the current implementation are only deferred, but immediate rules are needed, especially when introducing external asynchronous events and time events.

References

[Bou94] Bouzeghoub M., *Active Database Design*, Comet Seminar, 1994.

[Cat94] Cattel R.G.G.: *The Object Database Standard: ODMG-93, Release 1.2,* Morgan Kaufmann Publishers, Inc., 1994.

[CD91] Chakravarthy S., Mishra D.: *An Event Specification Language (Snoop) For Active Databases and its Detection.* UF-CIS Technical Report TR-91-23, September 91.

[CH90] Cary M., Haas L.: *Extensible Database Management Systems,* SIGMOD Record, v.19 n.4, December 1990, p.54-60

[CK+94] Chakravarthy S.,Krishnapsad V., Anwar E., Kim S.K.: *Composite Events for Active Databases: Semantics, Contexts and Detection*, Proceedings of the 20th VLDB conference, Santiago, Chile, 1994.

[CW96] Ceri S., Widom J.: *Active Database systems*, Morgan Kaufmann Publishers, INC 1996

[Day89] Dayal U.: *Queries and Views in an Object-Oriented Data Model*, Proceedings of the 2nd International Workshop on database Programming Languages, Glenden Beach, Oregon, USA, June 1989.

[DGA96] Dittrich K., Gatziu S., Geppert A.: *The Active Database Management System Manifesto: A Rulebase of ADBMS Features,* ACT-NET Consortium, 1996.

[Fah94] Fahl G..: *Object Views of Relational Data in Multidatabase Systems,* Licentiate Thesis LiU-Tek-Lic 1994:32, Linköping University, Linköping, June 1994.

[FaR97] Fahl G., Risch T.: *Query processing over object views of relational data,* to appear in VLDB journal 1997.

[FA+89] Fishman D.H., Annevelink J., Chow E., Connors T., Davis J. W., Hasan W., Hoch C. G., Kent W., Leichner S., Lyngbaek P., Mahbod B., Neimat M.A., Rish T., Shan M.C., Wilkinson W. K.: *Overview of the Iris DBMS*, in Kim W., Lochovsky F. H.: *Object-Oriented Concepts, Databases, and Applications,* ACM Press, Addison-Wesley, 1989, p.219-250.

[FKR95] Flodin S., Karlsson J. S., Risch T., Sköld M., Werner M.: *AMOS.v1 System Manual*, EDSLab internal report, 1995

[Flo96] Flodin S.: *Efficient Management of Object-Oriented Queries with Invertible Late Bound Functions,* Licentiate Thesis LiU-Tek-Lic 1996:03, Linköping University, Linköping, February, 1996.

[FR96] Flodin S., Risch T.: *Processing Object-Oriented Queries with Invertible Late Bound Functions*, Proceedings of the 1995 Conference on Very Large Databases, September 1996, p. 335-344.

[FRS93] Fahl G., Risch T., Sköld M.: *AMOS - An Architecture for Active Mediators*, NGITS'93, Haifa, Israel, June 1993.

[FSR93] Fabret F., Simon E., Regnier M.: *An Adaptive Algorithm for Incremental Evaluation of production Rules in Databases*, Proceedings of the 19th VLDB Conference, Dublin, Ireland 1993.

[GD93] Gatziu S., Dittrich K. R.: *Events in an Active Object-Oriented Database System*, Proceedings of the 1st Intl. Workshop on Rules in Database systems, Edinburgh, August 93.

[GJS92] Gehani N.H., Jagadish H.V., Shmueli O.: *Composite Events Specification in Active Databases: Model Implementation*, Proceedings of the 18th VLDB Conference, Vancouver, British Columbia, Canada, 1992.

[GM95] Gupta A., Mumick I. S.: *Maintenance of Materialized Views Problems, Techniques, and Applications*, Bulletin of the Technical committee on Data Engineering, Vol. 18 No. 2, IEEE Computer Society, June 1995.

[HD91] Harrison J. V., Dietrich S.W.: *Condition Monitoring in an Active Deductive Database*, ASU Technical report, TR-91-022, Department of Computer Science Engineering, Arizona, State University, Tempe, AZ. USA

[HW93] Hanson E., Windom J.: *An Overview of Production Rules in Database Systems,* The knowledge Engineering Review, vol. 8 no. 2, pages 121-143, June 1993.

[Kar95] Karlsson J.S.: *An Implementation of Transaction Logging and Recovery in a Main Memory Resident Database system*, Master Thesis LiTH-IDA-Ex-94-04, Department of Computer and Information Science, Linköping university, Linköping, June 1994.

[KF+95] Karlsson J., Flodin S., Orsborn K., Risch T., Sköld T., Werner M.: *AMOS.v1 User's Guide*, EDSLab internal report, 1995

[LR92] Litwin W., Risch T.: *Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates*, IEEE Transactions on Knowledge and Data Engineering, v.4 n.6, December 1992, p.517-528.

[Lyn91] Lyngbaek P.: *OSQL: A Language for Object Databases*, HPL-DTD-91-4, Hewlett-Packard Company, January 1991.

[Mel95] Melton J.: ANSI SQL Papers SC21 N9467, ANSI SC21 Secretariat, New York, U.S.A., 1995.

[Ris89] Risch T.: *Monitoring Database Objects, Proc.* VLDB conf. Amsterdam 1989.

[Shi81] Shipman D.W.: *The functional Data Model and Data Language DAPLEX*, ACM TODS, v. 6, n. 1, March 1981, p.140-173.

[Skö94] Sköld M.,: *Active Rules based on Object Relational Queries -Efficient Change Monitoring Techniques*, Licentiate thesis No. 452, Dept. of Computer and information Science, Linköping University, 1994.,

[SM96] Stonebraker M., Moore D.: *Object-Relational DBMSs: The next Great Wave*, Morgan Kaufmann Publishers, Inc., 1996.

[SR96] Sköld M., Risch T: *Using Partial Differencing for Efficient Monitoring of Deferred Complex Rule Conditions*, Proceedings of the 12th International Conference on Data Engineering (ICDE'96), New Orleans, Louisiana, February 26 - March 1, 1996, 392-401.

[SRF95] Sköld M., Risch T., Falkenroth E.: *Rule Contexts in Active Databases - A mechanism for dynamic Rule Grouping*, Proc. RIDS'95, athens, Greece, 1995.

[WF90] Widom J., Finkelstein S. J.: *Set-Oriented Production Rules in Relational Database Systems*, Proc. of 1990, ACM-SIGMOD Conference, p. 259-270.