Linköping Studies in Science and Technology
Dissertation No. 494

Active Database Management Systems for Monitoring and Control

Martin Sköld

Department of Computer and Information Science
Linköping University, Linköping, Sweden

Linköping 1997

Abstract

Active Database Management Systems (ADBMSs) have been developed to support applications with detecting changes in databases. This includes support for specifying *active rules* that *monitor* changes to data and rules that perform some *control* tasks for the applications. Active rules can also be used for specifying constraints that must be met to maintain the integrity of the data, for maintaining long-running transactions, and for authorization control.

This thesis begins with presenting case studies on using ADBMSs for monitoring and control. The areas of Computer Integrated Manufacturing (CIM) and Telecommunication Networks have been studied as possible applications that can use active database technology. These case studies have served as requirements on the functionality that has later been developed in an ADBMS. After an introduction to the area of active database systems it is exemplified how active rules can be used by the applications studied. Several requirements are identified such as the need for efficient execution of rules with complex conditions and support for accessing and monitoring external data in a transparent manner.

The main body of work presented is a theory for incremental evaluation, named *partial differencing*. It is shown how the theory is used for implementing efficient rule condition monitoring in the AMOS ADBMS. The condition monitoring is based on a *functional model* where changes to rule conditions are defined as changes to functions. External data is introduced as *foreign functions* to provide transparency between access and monitoring of changes to local data and external data.

The thesis includes several publications from both international journals and international conferences. The papers and the thesis deal with issues such as a system architecture for a CIM system using active database technology, extending a query language with active rules, using active rules in the studied applications, grouping rules into modules (rule contexts), efficient implementation of active rules by using incremental evaluation techniques, introducing foreign data into databases, and temporal support in active database systems for storing events monitored by active rules. The papers are complemented with background information and work done after the papers were published, both by the author and by colleagues.

Preface

Thesis Outline

This thesis is based on several conference and journal papers published during a period of four years. Each chapter usually contains background information for one or two papers and work done after the publications. A list of the papers can be found in the next section and the actual papers can be found in the last chapter of the thesis.

Chapter 1 presents the main differences between active database systems and “passive” database systems, and compares active database systems with other rule-based systems.

Chapter 2 presents the areas of Computer Integrated Manufacturing (CIM) and Telecommunication Networks and how database systems can be used to support different functions in them. Paper I is presented as a system architecture for a CIM system that uses active database technology.

Chapter 3 gives an overview of Active Database Management Systems (ADBMSs) and introduces the AMOS ADBMS. The active rules in AMOSQL are presented in Paper II along with changes and extensions made since the paper was published.

Chapter 4 discusses heterogeneous data management in the applications studied in chapter 2. The chapter also briefly presents the area of heterogeneous database systems and with the heterogeneous database architecture of AMOS presented in Paper III.

In chapter 5 Paper IV is presented as work on applying active database technology on a specific application, or more specifically using AMOS in the CIM architecture presented in Paper I. In Paper V a technique for organizing rules by grouping them into *rule contexts* is presented. Several scenarios for using active rules in CIM and Telecommunications are also presented.

Chapter 6 presents work on efficient execution of active rules. This chapter is based on Paper VI which presents a technique for incremental evaluation of rule conditions, *partial differencing*. Chapter 6 also presents a comparison between event propagation and incremental condition evaluation.

Chapter 7 discusses implementation issues of different aspects of the active rules and specifically the management of the *propagation network* and the *propagation algorithm* used for partial differencing.

In chapter 8 time series are presented for storing event histories in temporal functions that are monitored by active rules. The area of temporal and scientific databases are also briefly presented. Paper VII presents a technique for automatically building secondary indexes on time series.

Chapter 9 presents the concept of *foreign data sources* as a way to access

data originating outside the database. Different systems and protocols for accessing foreign data sources are discussed. Techniques for monitoring changes to foreign data sources and possible extensions to AMOS are also presented.

Chapter 10 concludes with a summary of the contributions in this thesis and presents possible future research directions.

In the appendix (chapter 14) the complete syntax of the active rules in the AMOS system is presented and the relation between Datalog and the relational operators. A formal justification for partial differencing is also presented.

In chapter 15 the different papers that the thesis is based on are presented.

List of Papers

Here follows a list of the published papers that this thesis is based on. The author has been a major contributor to and editor of all papers except Paper III and Paper VII.

Paper I

P. Loborg, P. Holmbom, M. Sköld, and A. Törne: A Model for the Execution of Task Level Specifications for Intelligent and Flexible Manufacturing Systems, in Proceedings of the Vth International Symposium on Artificial Intelligence, ISAI92, Cancun, Mexico, December 7-11, 1992. Also published in Journal of Integrated Computer-Aided Engineering (special issue on AI in Manufacturing and Robotics).

Paper II

T. Risch and M. Sköld: Active Rules based on Object-Oriented Queries, in special issue on Active Databases in Data Engineering Bulletin 15(1-4), Pages 27-30, 1992.

Paper III

G. Fahl, T. Risch, and M. Sköld: AMOS - An Architecture for Active Mediators, in Proceedings of the Workshop on Next Generation Information Technologies and Systems (NGITS'92), Haifa, Israel, June 1993.

Paper IV

P. Loborg, T. Risch, M. Sköld, and A. Törne: Active Object-Oriented Databases in Control Applications, in Proceedings of the 19th Euromicro Conference, Barcelona, September 1993.

Paper V

M. Sköld, E. Falkenroth, and T. Risch: Rule Contexts in Active Databases - A Mechanism for Dynamic Rule Grouping, in the Second International Workshop on Rules in Database Systems (RIDS'95), Athens, Greece, September 25-27, 1995, Springer Lecture Notes in Computer Science, ISBN 3-540-60365-4,

Pages 119-130, 1995.

Paper VI

M. Sköld and T. Risch: Using Partial Differencing for Efficient Monitoring of Deferred Complex Rule Conditions, presented at the 12th International Conference on Data Engineering (ICDE'96), New Orleans, Louisiana, USA, February 1996.

Paper VII

L. Lin, T. Risch, M. Sköld, and D. Badal: Indexing Values of Time Sequences, presented at the Fifth International Conference on Information and Knowledge Management (CIKM'96), Rockville, Maryland, USA, November 12-16, 1996.

Financial Support

This work has been supported by NUTEK (The Swedish National Board for Industrial and Technical Development), TFR (The Swedish Technical Research Council), CENIIT (The Center for Industrial Information Technology), and the ISIS project (Information Systems for Industrial Control and Supervision), Linköping University.

Acknowledgements

I would like to thank my supervisor Professor Tore Risch for his continuous support and for introducing me to the area of active database systems. Tore brought the WS-Iris system with him from HP-labs and his system has now become the AMOS system. Because of this my research got a running start.

I also would like to thank all the other members of the lab for Engineering Databases and Systems (EDSLAB) for inspiration and for fruitful discussions on the development of the AMOS system.

I also would like to thank the people in the robotics and measurement group at the Department of Physics and Measurement Technology (IFM), the staff at Ericsson Development, and the staff at Telia Research for providing equipment and sharing their knowledge.

Finally I would like to thank my family and friends for all their support over the years of my research studies.

To my parents for love and encouragement.

Martin Sköld

Linköping, July, 1997

Table of Contents

1 Introduction

1.1 Database Management Systems (DBMSs)

A Database Management System (DBMS) [39] is a general information management system that can manage many different kinds of data, stored in the *database*. By DBMS we here mean more than just a system that manages variables or files of data. The data can be both application data of different *types* and meta-data used by the DBMS to define the database layout, the database *schema*. In relational DBMSs (RDBMSs) data is defined with relations between data which are stored in *tables* and can be accessed through logical queries, or *relational views*.

The DBMS provides support for *logical views* of data that are separate from the physical views, i.e. how the data is actually stored in the database. This separation is accomplished by allowing applications to define, access, and update data through a Data Definition Language (DDL) and Data Manipulation Language (DML) combined into a *declarative query language* such as the relational query language SQL [6].

The DBMS provides *persistency* of data by ensuring that no data is lost in the case of system failures. The persistency can be achieved in many ways, e.g. by storing data and a log of uncommitted changes to data on disk.

DBMSs are traditionally self-contained systems (servers) and users or applications (clients) execute in separate processes (often on separate machines) and are provided access to the DBMS through a *client/server interface*. For many technical applications the performance requirements forces a tighter integration between the application and the DBMS. The DBMS is then *embedded* with the applications and executes in the same process (or at least shares the same address space). Applications with embedded DBMSs are provided fast access to data through a *fast-path interface*.

Another important aspect of functionality supported by a DBMS is *transaction management*. Transactions provide a mechanism for organizing and synchronizing database operations. Different users and applications can use transactions for defining sequences of database operations without, more or less, having to consider possible interaction with other users and applications. If, for some reason, something goes wrong during a transaction, the user or the application can choose to *abort* the transaction and all the database operations are undone. If a transaction is finished successfully, the DBMS can *commit* the transaction and make all the changes in the database permanent.

In recent years there has been development of DBMSs with more data mod-

elling support. This is often needed in technical and scientific applications where the schemas can be highly complex. In Object Oriented Database Management Systems (OODBMSs) OO programming languages have been extended to support database management through persistent object classes. A standard query language, OQL [20], has been defined for declarative access to the data. The OODBMS model does not, however, provide a fully declarative query language for both defining, accessing, and updating data (object definitions with attributes and methods are still defined in the OO programming language). OO programming languages such as C++ allow low-level operations on objects which makes the separation between a logical and physical view of data in OODBMSs difficult to support.

In Object Relational Database Management Systems (ORDBMSs) [118] the data models from the relational DBMS and the OODBMS have been merged. ORDBMSs provide declarative OO query languages such as OSQL [85] and SQL3 [90] for defining, accessing, and updating objects. Object classes are defined as types, and object attributes and methods are defined as functions. Functions are also equivalent to tables (or table attributes) and views in the relational model. The tables themselves are sometimes considered as a special abstract type [10], but which is accessed through functions.

1.2 Active DBMSs (ADBMSs) versus Passive DBMSs

Traditional DBMSs are *passive* in the sense that they are explicitly and synchronously invoked by user or application program initiated operations. Applications send requests for operations to be performed by the DBMS and wait for the DBMS to confirm and return any possible answers. The operations can be definitions and updates of the schema, as well as queries and updates of the data. Active Database Management Systems (ADBMSs) are event driven systems where operations such as schema changes and changes to data generate events that can be monitored by *active rules*. An ADBMS can be invoked, not only by synchronous events that have been generated by users or application programs, but also by external asynchronous events such as changes of sensor values or time. When monitoring events in a passive database, a *polling technique* or *operation filtering* can be used to determine changes to data. With the polling method the application program periodically polls the database by placing a query about the monitored data. The problem with this approach is that the polling has to be fine-tuned so as not to flood the DBMS with too frequent queries that mostly return the same answers, or in the case of too infrequent polling, the application might miss important changes of data. Operation filtering is based on the fact that all change operations sent to the DBMS are filtered by an application layer that performs the situation monitoring before sending the operations to the DBMS. The problem with this approach is that it greatly limits the way rule condition evaluation can be optimized. It is desirable to be able to specify the conditions to monitor in the query language of the DBMS. By checking the conditions outside the database the complete queries representing the conditions will have to be sent to the DBMS. Many DBMSs allow

precompiled *stored procedures* that can update the database. The effects of calling such a procedure cannot be determined outside of the database.

If the condition monitoring is used to determine inconsistencies in the database, it is questionable whether this should be performed by the applications, instead of by the DBMS itself. In an integrated ADBMS condition monitoring is integrated into the database. This makes it possible to efficiently monitor conditions and to notify applications when an event occurred that caused a rule condition to become true and that is of interest to the application. Monitoring of specific conditions, represented as database queries, can be performed more efficiently since the ADBMS has more control of how to evaluate the condition efficiently based on knowledge of what has changed in the database since the condition was last checked. It also lets the ADBMS perform consistency maintenance as an integrated part of the data management.

Internal ADBMS functions that can use data monitoring includes, for example, *constraint management*, *management of long-running transactions*, and *authorization control*. In constraint management, rules can monitor and detect inconsistent updates and abort any transactions that violate the constraints. In some cases compensating actions can be performed to avoid inconsistencies instead of performing an abortion of the complete transaction. In the management of long-running transactions, rules can be used to efficiently determine synchronization points of different activities and also whether if one transaction has performed updates that have interfered with another [32]. This can be used, for example, in cooperation with *sagas* [51] where sequences of committed transactions are chained together with information on how to execute compensating transactions in case of a saga roll-back. Groups of rules can be associated with a saga to detect any interference with the operations in the saga and that can redo the operations or roll-back the saga to undo the operations. In authorization control rules can be used to check that the user or application has permission to do specific updates or schema changes in the database.

Applications which depend on data monitoring activities such as CIM, Telecommunications Network Management, Medical [66] and Financial Decision Support Systems [26] can greatly benefit from integration with ADBMSs.

1.3 Using Rules as a Complement to Traditional Coding

Active rules can serve as a complement to traditional coding techniques where all the functionality of the system is specified in algorithms written in modules and functions. Active rules provide a more dynamic way of handling new situations and are often better alternatives to modifying old functions to cope with new situations. Great care has to be taken, however, when using active rules to avoid introducing unanticipated behaviour into the system. Misuse of rules, such as using too many levels of rules that can affect each other in unpredictable ways or attempts to use rules where traditional functions are more suited is one reason why the use of active rules in software development has only had limited success. A common technique that is used is to use rules for specifying parts of the system during the design phases and to use these rules as guidelines

for the actual coding phases or to compile the rules into corresponding functions to simplify the coding. This last technique is sometimes found directly supported in some programming languages such as Eiffel [88] where pre- and post-conditions on data can be specified. If the conditions are violated an error is generated. Rules can, however, specify pre- or post-conditions that should apply in many different situations not just in one piece of code. The rules can signal to the user or some application that a condition has been violated. Rules can also specify actions to be taken, such as removing inconsistencies by changing illegal values of data.

In most programming languages and query languages such as SQL3 [90] fault or signal handlers can be defined that catches error *signals*. Rules in an ADBMS can be seen as having similar behaviour, but catches database *events* such as updates.

Rules can also be used for monitoring changes to data. These are often specified as conditional expressions (*if-then-else*, or *case* expressions) in traditional coding. These are static expressions that cannot be changed unless the code is changed and recompiled. In databases that support incremental recompilation of functions and rules (such as the AMOS ADBMS), the rules can be dynamically changed. New situations can also be monitored by adding new rules.

1.4 Rule-Based Systems and Active Database Systems

In rule-based systems the rules can be used for different purposes. In fig. 1.1 the distinction is made between using rules for *monitoring*, *control*, and *reasoning*. We here make a distinction between *active database systems* [92] and other rule based systems such as *reactive systems* [87] (sometimes called real-time expert systems) and *knowledge-based systems* [68] (often just referred to as expert systems).

Active database systems are primarily database management systems with the main task of storing large amounts of data and providing efficient access to this data through a query language. In active database systems the rules are primarily used for monitoring changes to the data stored in the database. In reactive systems the rules are used for reacting to changes of some external environment and performing actions on (controlling) the environment in response to the changes. In knowledge-based systems the rules are usually used for reasoning using stored facts and by deducing new facts by using the rules. As can be seen in fig. 1.1 there is no sharp distinction between the three different kinds of rule systems. An active database system can do limited reasoning by using rules with more complicated rule conditions and which store new data in the database as new facts that signify that the rules have triggered. Control of the environment represented by the data in the database itself can also be performed, e.g. with constraint rules that modify the database to remove any inconsistencies. By allowing the active database manager to access an external environment that can be both accessed and updated, the rules in the active database can be used for control of an external environment as well. The primary use of active rules in an active database system as presented in this thesis is to

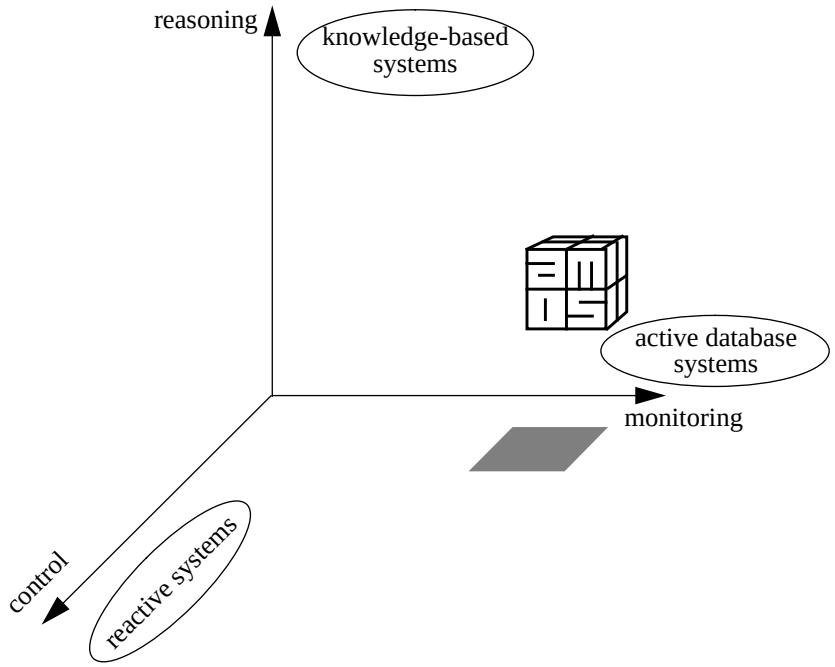


Figure 1.1: The relation between active database systems and other rule-based systems

monitor changes to the data that can be accessed in the database.

1.5 DBMSs in Large Complex Systems

DBMSs provide support for handling information in large complex systems. Integrating a system with a DBMS provides shorter development times (assuming the DBMS is already available), reduced complexity, reliability, and support for extensibility.

When complex systems are designed, the information or data that they should handle has to be considered early in the design process. Data modelling is often performed at an early design phase together with functionality modelling. In this phase specific modelling techniques such as Object-Oriented data modelling are often used. In later phases the actual data structures are chosen for storing the data. An OODBMS can support data modelling and select efficient data structures already provided by the OODBMS. If an RDBMS (non-OO) is used, then any OO models have to be translated into tables and views. Inheritance of attributes in tables and views will then have to be handled outside the DBMS. DBMSs usually support a separation between the logical view of the data seen by the system functions and the physical view, i.e. what data structures are used to store the data physically. This makes it possible to change

the data structures in the database without affecting the applications.

Large complex systems usually consist of many functions that all call for data management. Having a DBMS that can support the functions with this, the complexity of the system can be reduced. For example, if two functions use similar data structures, there is no need to implement these data structures for each function. By implementing them using abstract data types or object classes they can be reused by each function. In ORDBMSs [118] abstract data types are provided through an Object-Oriented extension of the relational model.

Another problem is that the reliability of these systems is often dependent on no data being lost, i.e. even if the system fails due to power loss or faulty hardware, the data should be available again as soon as the system recovers. This is a common trait in both Telecommunication Systems and many Computer Integrated Manufacturing (CIM) systems. A solution to these problems has been to introduce a DBMS into the system that supports persistency of data and transactions for organizing database operations.

A common problem in designing large complex systems is that the systems must support modification without too much redesign. Often it must be possible to modify data structures without recompiling application programs and sometimes even without taking the system out of service. It could be the case that some data structures are too small or perhaps lack some data fields that are needed to support new functionality. Such modifications are usually directly supported by a DBMS.

1.6 ADBMSs in Large Complex Systems

The introduction of a DBMS into a system provides a good platform for designing rules that access data from different parts of the system. Rules are dependent on the fact that all the information that is needed to check the rules is available. In a system without a general mechanism for storing data the rules have to be compiled into each module or function that can affect the rule condition. This limits the rule to just relating to data available in that module or function.

In an ADBMS active rules are managed by the ADBMS and the rules can thus directly access data stored in the database. Rules specified in a database can have conditions that span over data belonging to several modules of the system. The active rules can be used for directly supporting various applications with monitoring of changes to data in the database, with synchronizing activities in the system, and with maintaining the integrity of data in the database.

Care has to be taken when designing these systems to not introduce unwanted or unspecified communication between modules through the database. A common and successful technique in designing large systems has been to carefully design the interaction between different modules or processes by special interfaces, i.e. by exported interface functions or by inter-process communication. The design phase now has to take into consideration what data that

is going to be stored in the database for each module and what data is going to be visible to other modules.

By storing information about the state of the system in the database, e.g. the state of different hardware and software components, active rules can be used to monitor the state of the system itself. If the system is interacting with some external environment, e.g. a telecommunication network or a manufacturing plant, state information of these environments can be made available in the database as well. This could be done by mapping sensor data into the database and making it available in queries and rules. This does not have to mean that the sensor data is always stored permanently in the database. It may be the case that the sensor data is available to read as if it was stored directly in the database and that the ADBMS is informed when the sensor data changes. In many cases it makes no sense to store the data permanently since it changes quite frequently. The sensor data can sometimes be stored for logging purposes, but this might already be done in some other system that is part of the external environment. Allowing access to the state of the external environment through the database makes it possible to use active rules to monitor changes in the external environment.

1.7 Summary of Contributions

This thesis presents some case studies from the application areas of Computer Integrated Manufacturing (CIM) and Telecommunication Network Management (TNM). These application studies serve as requirements for the design of the AMOS ADBMS and especially the active functionality of AMOS that is presented in this thesis. The thesis is based on several publications such as papers at international conferences and articles in international journals. The major contributions within the field of active database systems are:

- Identifying the need of ADBMSs through the *case studies* of CIM and TNM. In the application studies the requirements for efficient execution of rules with complex conditions and the need for transparent access of external data were identified.
- Using active rules for *monitoring* and *control* in CIM and TNM.
- Identifying the need for *mediators* in CIM and TNM.
- Defining an ADBMS architecture.
- Identifying the need for generalizing the architecture towards *active mediators*.
- Adding active rules to an Object-Relational DBMS.
- Integrating (E)CA-rules into a query language.
- Rule modularization by grouping rules into *rule contexts.*
- Efficient rule evaluation techniques based on *partial differencing.*
- Defining external data in a transparent manner through the concept of *foreign*

data sources.

- Defining external events through the concept of *foreign events* of *foreign functions.*
- Work on introducing *time series* for storing event histories.
- Work on new *indexing techniques* for inverse queries over time series.

2 Background

2.1 Application Studies

This chapter presents application studies done as background research to find what requirements there are on an ADBMS for supporting various technical applications. The goal is to provide the ADBMS with general functionality that is suitable for the various needs of different applications. Some functionality might not be as important for one application as for another, but all functionality should be as general as possible instead of implementing very specialized functionality tailored for just one specific application.

Two application areas were studied and are presented in this thesis:

- Computer Integrated Manufacturing (CIM)
- Telecommunication Networks

2.2 Computer Integrated Manufacturing (CIM)

Computer Integrated Manufacturing (CIM) is a broad term that covers all aspects of automated manufacturing from using welding robots in car manufacturing to using specialized equipment for making integrated circuits or controlling a steel- or paper-mill. There are usually many computer systems used in manufacturing plants and the number of systems and level of automatization is constantly increasing. Most systems that are considered are directly involved in controlling the manufacturing process, usually called *process control systems*. These systems *control* a manufacturing process using *actuators* (e.g. conveyor-belts, feeders, robots, lathes, or boilers) and *monitor* the progress using *sensors* (e.g. speedometers, position sensors, force sensors, image processing systems, thermometers, or pressure gauges).

Some other systems that are also sometimes covered by CIM include various systems that are being used within manufacturing companies. These can be systems involved in the design process such as product specification and Computer Aided Design (CAD) systems, systems that handle parts and products in stock, and economic information systems such as product costs and sales information. A current trend in many manufacturing companies is to integrate all of these systems to provide better control of the whole manufacturing process, not just the process control.

2.3 DBMSs in CIM

CIM systems handle many different kinds of information for controlling the manufacturing process such as product data (e.g. what parts a product consists of), parts data (e.g. physical data such as size, weight, and number of parts available in stock), data related to the manufacturing equipment (e.g. configuration data), sensor and actuator data. Other data that can be handled by a CIM system, but which is not directly used in the process control can be product specification data (e.g. CAD-drawings), economic data (e.g. product and part costs), and sales data (e.g. how many products have been sold and thus have to be manufactured). The types of CIM applications that can be considered as candidates for the use of (active) database technology are applications where a fairly large amount of data access is needed during the automated process. This could be data such as information about the components involved in an assembly, data about the machines involved and sensor data stored or data directly accessible in the database. The data could also be information about the number of components in stock. This often involves applications where the level of autonomy has to be high and thus allowing the CIM system to operate without too much human intervention. This could be in a system that is more fault tolerant, e.g. by using sensors to detect abnormal situations and to deal with them without an operator having to restart the system, and that can also interact with other systems, e.g. to automatically order more components when the stock is running low.

2.3.1 The ARAMIS Project

The ARAMIS project [83][123] was the continuation of a joint research project between the Department of Computer and Information Science (IDA) at Linköping University, ABB Corporate Research and ABB Robotics in Västerås, Sweden. The project continued as cooperation between IDA and the Department of Physics and Measurement Technology (IFM). The work at IDA consisted of developing the software platform for the target hardware (a real-time system and robot with a gripper and various sensors) being developed at IFM. The software platform was developed in a three-layered architecture. The layers are: the *task level*, the *control level*, and the *physical level*.

On the task level, task programs can be written that specify the main tasks of the application in a declarative rule-based language. The task programs are written using a graphical notation and using special programming tools. In the task-level programs low-level details can be ignored such as how the actual control algorithms will perform different high-level operations. The task programs operate on objects in a World Model (WM) which is stored in a database. Objects (called components) in the WM can be active, which means that if the task level changes attributes of active components, the WM can issue calls to the control level that executes algorithms that perform the corresponding changes to the physical object that is represented by the active component in the WM. The architecture is presented more thoroughly in Paper I.

This work uses an ADBMS for control of manufacturing equipment with obvious real-time requirements, but the focus was not on real-time databases [101]. One basic idea in the architecture is to push real-time requirements into the control algorithms as much as possible, i.e. out of the database and the active rules. The control algorithms can be cyclic operations with fixed cycle times that can be adjusted to meet hard real-time requirements. Some soft real-time requirements can still be present on the database.

The ADBMS should provide high-performance transaction processing through efficient rule/query processing and efficient updates of the database. To meet these requirements a main-memory DBMS [34][38][52] was considered as the most likely candidate.

My work consisted of developing the control software which included developing languages and tools for specifying active components that perform the control of the physical hardware (see Paper I). This work was the initial incentive to focus my research on the area of active database systems.

2.4 About Paper I

This paper presents the ARAMIS architecture with an emphasis on the control level and the specification of *active components*. The model chosen for defining components was based on Object-Oriented (OO) techniques. The components can be either *passive* or *active*. Passive components represent modules containing functions that have some common functionality such as specialized algorithms for 3D-rotation of objects. The inheritance structure of passive components represents *specialization* (or generalization) of functionality such as a 3D-rotation component can be defined as a specialization of a 2D-rotation component. Active components represent objects with a state and are used for representing objects in the real-world such as equipment in the manufacturing plant and the parts being assembled. The inheritance structure of active components represents an *is-a* (or instance-of) hierarchy. In [120] general definitions of different inheritance models can be found. The OO model chosen for the components later influenced the choice of using an OODBMS to represent the WM. The ARAMIS system had a primitive main-memory active database system that was used for storing the active components, but it lacked well-defined transactions and a query language. This database system was later substituted with the AMOS ADBMS [43]. AMOS is presented in chapter 3. In chapter 5 the use of an ADBMS in CIM applications is more discussed.

2.5 Telecommunication Networks

Telecommunication networks consist of the infrastructure and the equipment needed to provide different telephony services. Traditional telecommunication networks provide transfer of low bandwidth analog data such as voice data in a point-to-point manner. The services provided by the telecommunication network can be divided into services provided to the end user and services pro-

vided to the network operator. Traditional user services include Plain Ordinary Telephony Service (POTS), i.e. basic point-to-point voice-based communication without operator assistance, different subscriber services such as call transfer, call waiting, and number presentation of who is calling. Traditional operator services include monitoring network usage and billing subscribers, adding/removing subscribers, load balancing the network (e.g. transferring traffic from heavily used sections to less used sections, sometimes by splitting one high bandwidth connection into several connections), reconfiguring the network without disrupting network traffic, and network supervision functions (e.g. monitoring network overload and equipment failure). Future telecommunication networks will provide transfer of high bandwidth digital data in both point-to-point and in a broadcast (one to many) manner. Today's fixed networks are digital between the exchanges, but usually not all the way to the end users (subscribers).

In ISDN (Integrated Services Digital Networks) subscribers can be given a fixed medium bandwidth transfer. The basic idea in ISDN is that a digital *bit-pipe* through an Integrated ISDN Transport Network is set up between users (fig. 2.1). The bits can originate from any digital ISDN device such as a digital telephone, a digital fax, or a terminal (or any general computer). The connec-

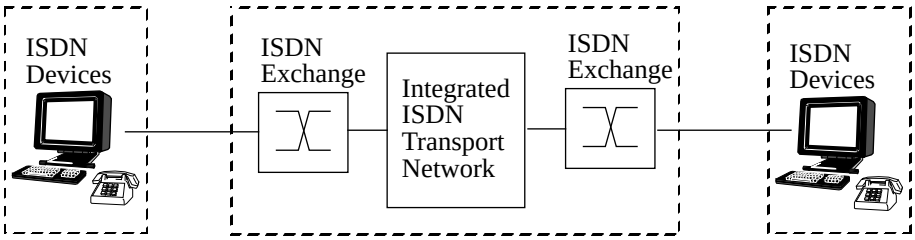


Figure 2.1: The basic ISDN network

tions to the end users are defined to use existing twisted pair connections using special ISDN interface hardware. Within the transport network any media can be used such as optical fiber cables. The ISDN bit-pipe supports multiple channels interleaved by time division multiplexing. Several channels have been defined:

A - 4 kHz analog telephone channel
B - 64 Kbit/sec digital PCM channel for voice data
C - 8 or 16 Kbit/sec digital channel
D - 16 or 64 Kbit/sec digital channel for out-of-band signalling
E- 64 Kbit/sec digital channel for internal ISDN signalling
H - 384, 1536, or 1920 Kbit/sec digital channel

Different combinations of these channels have been defined such as the basic rate 2B+1D which can be viewed as a replacement for the communication in POTS. The ISDN standard as it was initially defined has never been realized in

integrated large-scale public networks. ISDN is usually provided through special networks that work in parallel with the public telecommunication networks or in local networks through a Private Branch eXchange (PBX). A major reason why the ISDN standard has not been widely implemented is that many new applications require network performance above that which an ISDN network can provide. Applications such as transfer of images, video, or high-fidelity sound have a very *bursty* nature, i.e. low data transfer can be followed by sudden burst of high data transfers. The basic ISDN standard provides a statically allocated bandwidth and applications must allocate enough bandwidth to support the maximum bandwidth that they need. For bursty applications this leads to a lot of waste of bandwidth since the whole allocated bandwidth is only used parts of the time. To support these kind of applications the basic ISDN standard was extended and was named Broadband ISDN (B-ISDN). B-ISDN supports dynamic bandwidth allocation by using the ATM (Asynchronous Transfer Mode) technology to implement the Integrated B-ISDN Transport Network.

The ATM network standard has been defined to support integration of both local and public networks consisting of different transport media such as unshielded twisted pairs, shielded coaxial cables, and optical fibers using different kinds of broadband switches such as local ATM PBXs and large public ATM exchanges. The broadband ATM devices in an office or a home can be defined to belong to ATM workgroups that are connected to ATM PBXs in private (corporate/enterprise) networks or directly to local ATM exchanges in a public carrier's ATM network (fig. 2.2). The private ATM networks can consist of several ATM PBXs in a Local Exchange Carrier Network and the public ATM network can consist of several networks with ATM Exchanges in Inter Exchange Carrier Networks that are being managed by different network providers.

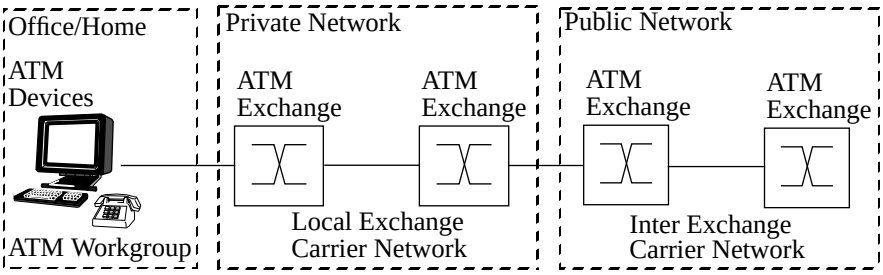


Figure 2.2: The ATM (B-ISDN) network main layout

In an ATM-network users can be given dynamic high bandwidth transfer. The physical communication layer is actually not part of the ATM specification, but standards for optical networks such as SONET/SDH (Synchronous Optical Network/Synchronous Digital Hierarchy) specify the speeds, 155.5 Mbit/sec, 622 Mbit/sec, 2.4 Gbit/sec. ATM networks transfer data as digital packages containing parts of the data along with the destination address and control data (e.g.

for error checking). The packages can be routed different ways depending on the current load situation. The exchanges disassemble the data from the sending party into a sequence of packages and perform *package switching* by routing the packages to their correct destination and assemble them in the correct sequence and send the data to the receiving party. How connections are set-up between the users and how the connections are controlled is different from how connections are managed in traditional telecommunication networks. The connections in ATM networks must have higher reliability and this will make both the control and management of these networks more complicated. In section 2.6.1 the control of these networks and how connections are set-up are discussed. In section 2.6.2 the management of these networks and set-up connections is discussed. The main difference to the users of the networks will be increased performance through a high bandwidth network, more user services, and the fact that the communication is digital all the way making modems redundant for digital data transfer.

Today's second generation mobile telecommunication networks, i.e. GSM/ TDMA[1] or CDMA[2] cellular phone networks [67], are already digital all the way (the first generation was analog), but do not provide very much bandwidth to the subscribers. When ATM-networks are widely available, cellular phone networks will probably be upgraded to benefit from the higher broad band capability, but there will probably still be a limitation on the bandwidth available in the mobile phone - base station connection because the radio band will always be cramped. In the Universal Mobile Telecommunications System (UMTS), defined in an EU RACE-program (Research on Advanced Communications), a third generation mobile telecommunications network has been defined. In UMTS the mobile network has been integrated with a broad-band package switched network such as ATM[3]. Mobile phones are already beginning to be integrated with hand-held computers to become *mobile workstations* [72]. Users with mobile terminals will thus be mobile and have access to broad band services (a bandwidth of 2 Mbit/sec. has been defined). Work is also in progress on *mobile ATM* [127] where users can access an ATM network directly from mobile terminals. One complication with using the ATM protocol all the way to the mobile users is that there are no sequence numbers in ATM packages. In a wireless network ATM packages can become misordered and there are proposals for adding sequence numbers to help reordering ATM packages at the receiving end [127].

Future user services other than POTS and the standard subscriber services (e.g. call transfer and call waiting) will include direct, real-time, transfer of any digital data (e.g. digital television and teleconferencing), and other services

1. In the Global System of Mobile communications (GSM) a Time Division Multiple Access is used for multiplexing several logical channels onto each physical carrier channel.
2. Code Division Multiple Access (CDMA) is a future North American mobile telecommunication system.
3. Initially UMTS was to be integrated with B-ISDN, but this is not considered a good technical solution anymore.

using non-real-time data transfers such as electronic mail, news services such as electronic newspapers and stock market information, accessing the Internet, and video-on-demand services. Monitoring services such as a service that allows users to directly monitor how much money he/she has spent might be possible. Future operator services might include better monitoring of network use and misuse (e.g. by using encryption and authorization control), controlling how much bandwidth is given to different users, billing according to used bandwidth, better support for load balancing the network (e.g. automatic splitting of high bandwidth connections through package switching and delaying transfer of non-real-time data until low traffic periods), better support for dynamic reconfiguration without disrupting network traffic (e.g. by having better support for rerouting data away from equipment that is being replaced or upgraded).

2.6 DBMSs in Telecommunication Networks

In the area of telecommunications there are many different needs for DBMS support. Telecommunication networks already have DBMSs integrated with them and will have even more so in the future. Telecommunication networks are large heterogeneous systems with many, sometimes conflicting, needs that the DBMSs must fulfil [58]. When discussing DBMSs in telecommunication networks it is important to separate between *network traffic control*, *network management*, and *network applications*.

The network traffic control involves the actual operation of the network in terms of setting up communication paths, maintaining them, and disconnecting them. In this thesis POTS (i.e. point-to-point communication) and standard subscriber services (e.g. wake-up call, call diversion, call waiting, malicious call tracing) are considered to belong to network traffic control. Network traffic control have requirements on DBMSs to provide high throughput and a large number of parallel transactions, main-memory storage, fast-path interfaces to programming languages, and real-time support. In network traffic control, availability and reliability is important [122], but losing a single connection is not a catastrophe.

Network management [59] involves monitoring network traffic in terms of performance (network throughput), fault management, and configuration management. Network management makes requirements on the DBMSs to provide support for interconnecting with other DBMSs and with other systems and to provide support for monitoring connections, alarms in the network, and network configuration changes. Collecting accounting data to support billing of network use is also a task for network management. The DBMSs for network management have a high requirement on reliability since losing, for example, accounting data is unacceptable to a network operator. The network management also requires support for more complex data models to support modelling the layout of the actual network (such as network elements and their connections) within the database. The DBMSs need to store accounting data securely on disk, but might still be main-memory based to meet some of the perform-

ance requirements and with a disk for backup only.

Network applications that use a DBMS can include applications other than traditional telephone calls such as Internet access (e-mail, news, WWW, file transfer, and remote system access), text and voice mail, multi-media, and video-on-demand. Network applications will require support from a high-performance DBMS that can handle a large number of simultaneous transactions. To meet these requirements main-memory DBMSs can be considered. Disk based DBMSs that can store large amounts of data will also be needed for logging purposes and for applications needing to store large volumes of data. Support for new data structures will be needed for storing, for instance, voice data, graphical data, and video data in the databases. To support queries over these new data structures the DBMSs must support efficient indexing techniques and optimization of queries that access them.

2.6.1 Telecommunication Network Traffic Control

Network traffic control is probably the area within telecommunications where there are the most manufacturer-specific solutions. Each telecommunication exchange (switch) developer has its own solutions of how to handle data. DBMSs are being integrated into the software platforms for the switching systems. In an ATM exchange the data being stored can be connection data, system management data, and configuration data.

Connections in an ATM network are associated with each other through *Virtual Circuit Identifiers* (VCIs) that are sent along with data packages to identify where they come from and where they should be routed. Two kinds of virtual circuits have been defined, *Permanent Virtual Circuits* (PVC) and *Switched Virtual Circuits* (SVC). PVCs require that the customer defines the characteristics of the connection, including the end-points. SVCs allow connections to be established on-demand between any two points in the network. SVCs are going to be needed in a dynamic public ATM network and are assumed in the continued discussion. When new connections are established, new VCIs are allocated and are maintained until the connection is terminated causing the VCI to be deallocated (and be reused by new connections).

Connections (trails) routed through several exchanges will have several VCIs for each sub-connections (segments) through the network. Each ATM exchange will keep records on how each incoming and outgoing sub-connection is connected by recording the associated VCIs. Each sub-connection will be monitored by the ATM-exchanges making sure that the connections get the good throughput by varying the actual bandwidth acquired during fluctuations (bursts) in the transmitted data. Data about setup connections, i.e. associated VCIs, can be stored in local databases. Network usage can be temporarily stored for each sub-connection, but will be forwarded to network management systems for monitoring the overall performance and for calculating the total network usage for billing. Maintaining the complete connections through the network is part of the network management.

System management such as monitoring the performance of the whole

exchanges, different physical links, and logging of alarms will need DBMS support. Such a DBMS must be accessible from or have direct contact with a network management center that monitors the larger part of the network that the exchange belongs to. System configuration data, such as hardware and software configuration, will probably be handled by a DBMS as well. System reconfiguration can be managed more securely by using DBMS transactions to atomically change many parameters simultaneously.

In local exchanges (with directly connected subscribers) a DBMS can be used to store subscriber information such as subscriber numbers, subscriber services, accounting information. For fast number analysis special data structures for fast look up and with possibly incomplete keys (parts of subscriber numbers) are usually implemented. The searching can usually start before the subscriber has dialled all the digits. Subscriber numbers are usually defined in number series according to a number plan that specifies country and area codes. These are usually defined and stored in a hierarchical structure that is searched for finding where incoming calls should be routed. Usually no single database contains all the numbers, but each exchange DBMS can determine where (in what other exchange) the number analysis should be continued. In future mobile telecommunication networks the subscriber numbers will be global, i.e. without any fixed association with where the subscriber is physically located. This will change how subscriber numbers are looked up in the database. Since mobile subscribers usually type in the whole number they are calling, there is no need for incomplete search keys. Some hierarchical definition will probably still be needed to avoid full replication of all numbers in the local exchanges.

The local exchanges usually have charging functions that monitor the number of time periods used in calls set-up by local subscribers. Local accounting information is usually stored temporarily before it is forwarded as charging records to some external DBMS in the network management system. In the future such charging will also include the use of network services not handled by the local exchanges. On-line billing of network services will probably be performed by DBMSs of the network management system.

Subscriber services usually have specific functions or modules in the system that need to store their own information in the database (such as to what numbers to transfer calls to subscribers that have activated the call diversion service). In Intelligent Network Services some new services such as routing calls to different locations at different times of the day, translation of numbers, and redirecting of charging have been defined. To support such services Service Control Points (SCPs) have been defined that will use network DBMSs for looking up data.

In third generation mobile telecommunication networks such as UMTS users will not be physically connected to a particular local exchange and will probably have portable subscriber numbers that can be used to find them anywhere in the network. Home Location Registers (HLRs) and Visited Location Registers (VLRs) are specialized DBMSs used in second generation mobile networks for finding the location (local home exchange and current location) of mobile users. In UMTS HLR/VLRs will probably use SCPs and network

DBMSs to support portable subscriber numbers.

In section 5.6.1 the use of ADBMSs in telecommunication network traffic control is discussed.

2.6.2 Telecommunication Network Management

Telecommunication networks are large hierarchical networks consisting of subscribers connected to local exchanges, local exchanges connected to transit exchanges, and network supervision centers that monitor the traffic between the exchanges. DBMSs for network management are integrated parts of these networks to some extent and will be even more so in the future [59].

The tasks of network management are the following:

- Performance Management.
  To monitor the status of network resources, traffic load, equipment utilization, and identify exceptional conditions.

- Fault Management.
  To detect alarms, diagnose problems, and apply control.

- Configuration Management.
  To provide support for installation of new equipment or services, audit, and reconfigure network resources.

- Accounting Management.
  To provide billing data, resource usage reports, and cost calculations.

- Network Planning Management.
  To prepare for capacity growth, contingency, and strategic planning.

- Security Management.
  To handle authorization and authentication.

Network management of ATM networks [5] is divided into several levels and interfaces. The ATM Forum is developing a five-layer ATM management model for Operation, Administration, and Maintenance (OAM) of ATM networks. The model defines interfaces for managing hybrid networks that consist of both private and public networks. OAM cells are being defined that automatically distribute management information throughout the ATM network. The model also includes end-to-end management based on the Common Management Information Protocol (CMIP).

Local networks (LAN) connected to an ATM network can use the ATM Data Exchange Interface (DXI) for exchanging data. Network management in private ATM networks has been defined by the Interim Local Management Interface (ILMI) which is based on the Simple Network Management Protocol (SNMP). SNMP was defined by the Internet Engineering Task Force (IETF) and is widely used in management of computer networks. SNMP is based on

the definition of Management Information Bases (MIBs) which support reading, writing, and monitoring changes to data related to network elements. The IETF has produced an ATM MIB for SNMP and the ATM Forum has defined the ATM DXI MIB (as an extension of the ISDN MIB). A remote monitoring AMON MIB (based on RMON, Remote MONitoring) has also been defined for support of more automatic ATM network monitoring. MIBs define objects (or variables) which can be polled to monitor the operation and performance of a managed component. Managed components can be any piece of hardware in the communication network. More discussions on SNMP and MIBs can be found in section 9.5.3.

The Customer Network Management (CNM) interface makes it possible for customers of an ATM service to manage certain aspects of the service from their own local network management system. The CNM is the interface between the customer and the carrier's public network management systems and gives the customer a view into the carrier's network. CNM systems also rely on MIBs for accessing data. The goal of the integration of customer and carrier's network management systems is that customers will have real-time control over the services they use. The carrier wants to provide the private network management with the ability to monitor and control the quality of the services received, but without giving away full control of the network.

To support management of the public networks the interfaces based on the Network Management Level (NML) views and the Element Management Level (EML) views have been defined [81]. The NML provides an abstraction of the functions provided by the systems that manage network elements on a collective basis (the network management systems) to make it possible to monitor and control the network end-to-end. The EML provides an abstraction of the functions provided by the systems which manage each network element on an individual basis (the network control systems). The basic idea behind the interfaces is to allow the network management system to work on an integrated and *logical view* of larger parts or the complete public network. More on logical views of telecommunication networks can be found later in this section.

Since there will probably be several carrier network providers, there is a need for an interface for integrating different carriers' network management systems. This is needed to provide monitoring of complete connections through the whole network. Information such as forwarding of network usage, billing information, and alarms will have to be forwarded using standard formats. This interface has yet to be defined.

In fig. 2.3 an overview of the different network management interfaces can be seen with the interface codes explained below.

- M1, M2 - Interim Local Management Interface (ILMI) based on the Simple Network Management Protocol (SNMP)
- M3 - Customer Network Management (CNM) interface
- M4 - Interface providing Network Management Level (NML) views and Element Management Level (EML) views of the public network

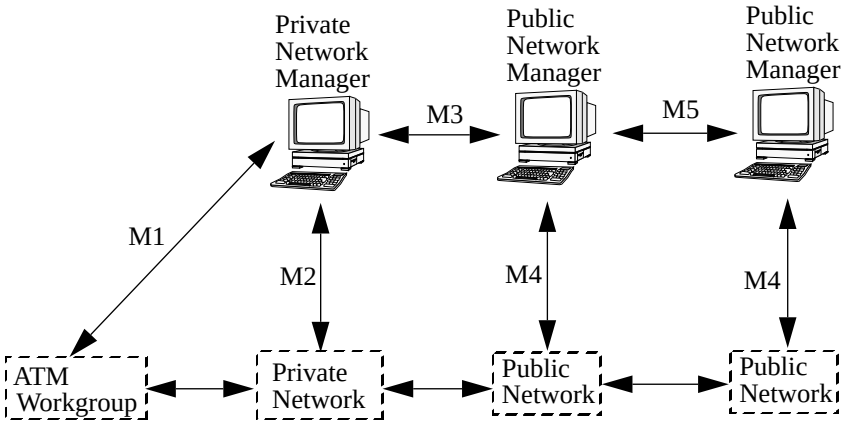


Figure 2.3: The ATM Forum Management Interface Reference Architecture

- M5 - Interface between the public network management systems

In telecommunication network management, logical views are usually defined in terms of the physical network (fig. 2.4). Logical names of devices and users will have to be translated to physical addresses by a name server function. The views reflect

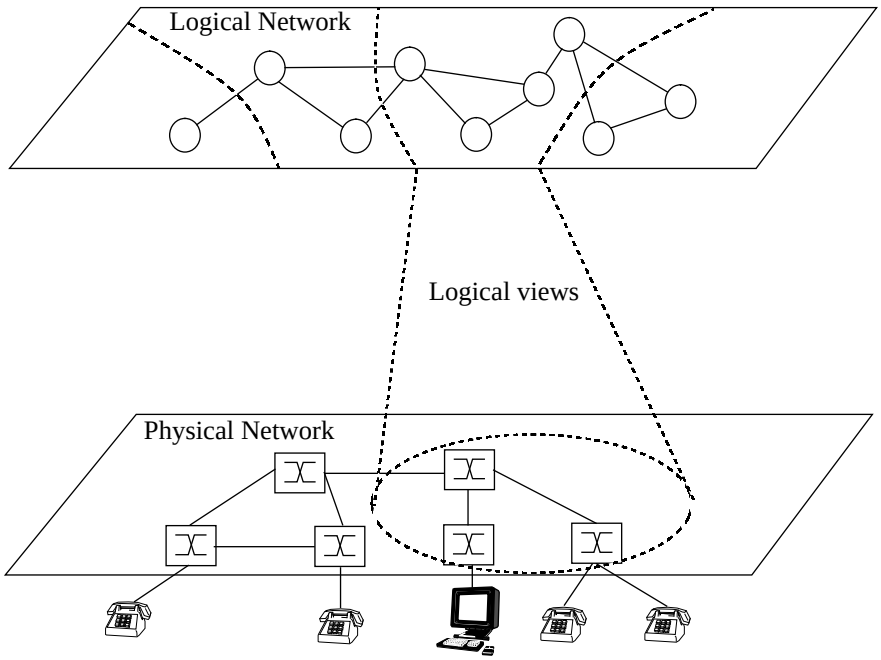


Figure 2.4: Mapping the physical network to a logical network through logical views

area code regions and geographical regions more than how the network is physically interconnected. The operators of network management centers will usually find it more convenient to access the different parts of the network using the logical view. The physical network addresses will usually only be needed when devices and users are added (removed) to (from) the network.

The views can be defined on several levels for local and regional network management (fig. 2.5). In reality (as for example is defined in the North Amer-

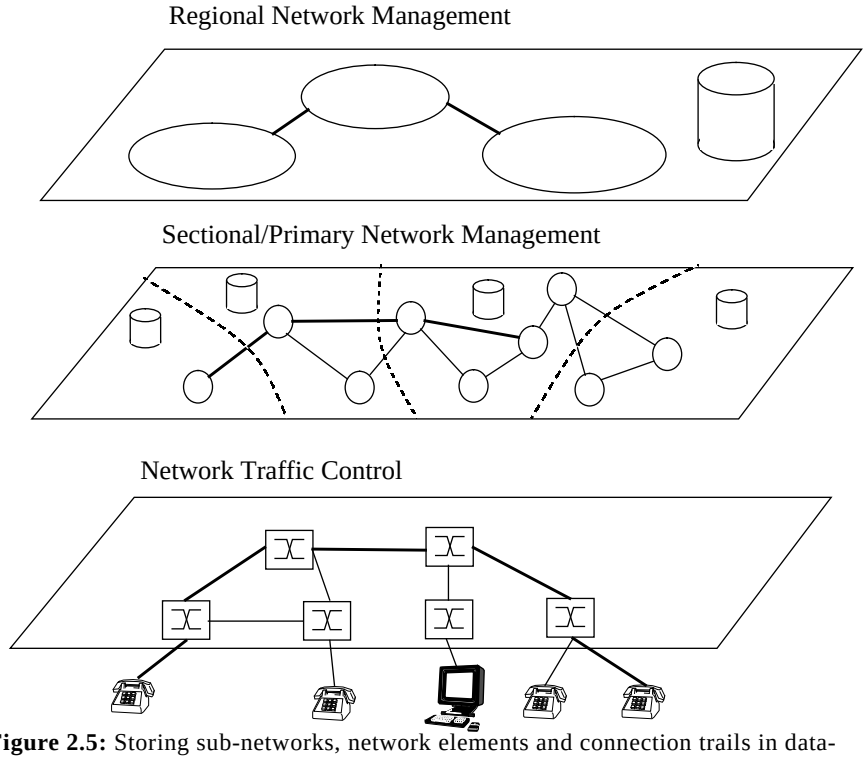


Figure 2.5: Storing sub-networks, network elements and connection trails in databases

ican Telephone Switching Office Hierarchy) the telecommunication networks consist of several levels such as regional centers, sectional centers, primary centers, toll centers and local offices. DBMSs will be needed on all these different levels. The information that will be stored in the databases includes static data such as objects representing the network elements, the network configuration, and dynamic data such as the status of the network elements, statistics, set up trail connections, actual network usage, and billing information. In telecommunication network management there is a greater need for data modelling than in network traffic control. The network modelling is likely to be defined using international standards based on the Object-Oriented (OO) paradigm.

The Guidelines for Development of Managed Objects (GDMO) [74] is a standard for modelling of network elements based on OO techniques. Network providers can define how their networks are logically connected using GDMO and then use the NML and EML views to define how the logical view of the network is mapped to the physical view, i.e. the relationship between how the elements of the logical network are defined to be interconnected and how the physical network elements are physically interconnected. The logical view makes it easier to understand how the network is connected and easier to manage by network operators. Operations such as monitoring can be done on the logical model with all the requests and data being translated between the physical and the logical views. Changes to the physical and the logical views and the mapping between them must be possible to allow for reconfiguration without taking the whole network out of service.

To support the network management the logical views can be stored in a database with direct support for the network modelling. OODBMSs, for example, can be used to directly store GDMO based models of the networks. Interconnections between DBMSs at different levels of the network hierarchy can provide the mappings between the different views. Support for defining the NML/EML mappings and doing the actual translations will have to be provided as part of the functionality of the network management software that is tightly integrated with the DBMS.

The DBMSs can be seen as being part of a heterogeneous system that connects different databases and network elements (fig. 2.6). The data sent between the different databases and the network elements can be:

- Alarms signalling different errors in the network. Failed network elements and traffic congestion.
- Reconfiguration information, new added network elements, removed network elements, and new interconnections between network elements.
- Information about set-up connections.
- Accounting information.

To allow databases to access other databases they have to be designed with this in mind. In *heterogeneous DBMSs* access to other databases is supported and queries spanning over several databases can be defined and optimized. In chapter 4 heterogeneous DBMSs are discussed further. To support access to other non-database sources of data such as network elements the DBMSs must support this as well. Standards have been defined such as different MIBs that specify how the different objects (data) in other databases and network elements can be accessed and monitored. DBMSs integrated in network management have to be designed to support these standards. In chapter 9 *foreign data sources* are defined and discussed that allow DBMSs to access and monitor changes to data that is not stored physically in the database.

In section 5.6.2 the use of ADBMSs in telecommunication network manage-

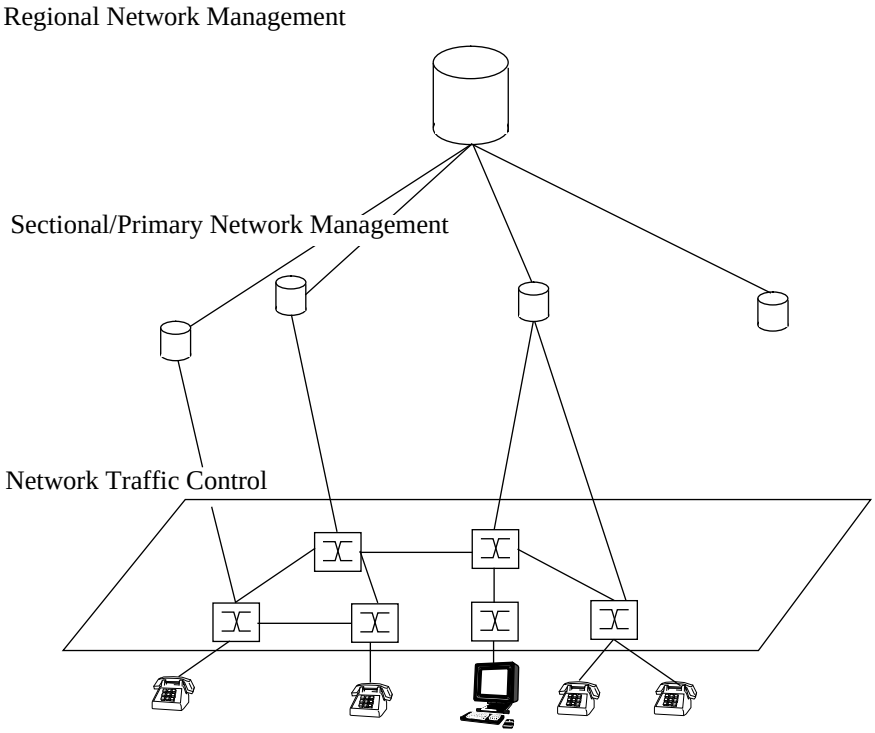


Figure 2.6: Connecting DBMSs and network elements in different levels as foreign data sources

ment is discussed.

2.6.3 Telecommunication Network Applications

Current Internet applications such as e-mail, news, and WWW (the World Wide Web) will most likely be provided through future telecommunication networks [15]. This can be achieved by just extending the Internet to partly run on top of the telecommunications networks through an IP (Internet Protocol) gateway in an ATM network (see section 9.5.3). It can also be achieved by introducing these applications as new telecommunication services separate from equivalent applications on the Internet. Future broad-band telecommunication networks can hopefully provide better bandwidth, reliability, and support for billing of used services which cannot be provided by the Internet today. Future applications such as multi-media e-mail, interactive TV (multi-media WWW), and video-on-demand can be provided directly to the telecommunication network users or indirectly from the Internet through an IP/ATM gateway.

Many of these applications will need DBMS support. DBMSs can be used as search engines, e.g. searching for a particular service, for storing multi-media

data, and for on-line billing of the services provided. These applications will require more support for storing non-tabular data, such as multi-media documents consisting of both audio and video information. BLOBs (Binary Large Objects) are used as a common term for these new data structures. Support for extending the databases with these data structures is not enough in itself; support is also needed for accessing these objects or parts of them using indexed search and through a query language. The DBMSs must support extensions of their type systems with new data types and with application-specific operations on them. These kinds of extensible database systems have been named *Object-Relational Database Systems* [118].

In future mobile telecommunication networks such as UMTS where users will have mobile terminals there are special challenges to application data management [71]. To support mobile applications DBMSs for *mobile computing* [72] have to deal with users who can connect to the database for brief periods and then disconnect while moving somewhere else. Data can be stored both in network databases and locally in the mobile terminal. Here a separation is made between *global* and *local* data management. Global data management deals with network problems such as locating, addressing, replicating, and broadcasting application data to users. Local data management refers to end-user level data management in the mobile terminal and includes energy-efficient data access with caching of data, management of disconnection and reconnection, and management of query processing for efficient navigation through the network to find the desired data. There are many new possible applications for mobile DBMSs. One interesting application is that of integrating *vehicle navigation systems* and mobile telecommunication systems. In vehicle navigation systems the positions of vehicles are monitored using *GPS (Global Positioning System)* [112] which is based on using satellites together with the reference signal from mobile telecommunication base stations[1]. Mobile terminals can be used for sending and receiving data related to the position of the vehicle. This application will rely heavily on DBMS support for storing and sending information requested by the user such as maps and multi-media data related to the vehicle position.

In section 5.6.3 the use of ADBMSs in telecommunication network applications is discussed.

1. For improved accuracy in determining a more exact position compared to the position provided by only using GPS.

3 Active Database Management Systems

3.1 An Overview of Active Database Management Systems (ADBMSs)

In System R [6] a *trigger* mechanism was defined that could execute a pre-specified sequence of SQL statements whenever some triggering event occurred. The triggering events that could be specified included retrieval, insertion, deletion, and update of a particular base table or view. Triggers had immediate semantics, i.e. they were executed immediately when the event was detected. In System R it was also possible to make *assertions* that specified permissible states or transitions in the database through *integrity constraints* that always had to be true after each transaction. Specific events had to be specified for when assertions were to be checked in the same way as with triggers. Assertions usually had deferred checking semantics, i.e. they were checked when transactions were to be committed. If an assertion failed, then the transaction was aborted.

The term *active databases* was coined in [92] as meaning “a paradigm that combines aspects of both database and artificial intelligence technologies”. In [92] a mechanism for constraint maintenance, *Constraint Equations*, was presented as a declarative representation for a set of related Condition-Action rules.

In HiPAC [23][29][31][133] a thorough specification was made of what different mechanisms are desirable in an *Active Database Management System* (ADBMS). Active rules are defined as *Event-Condition-Action* (ECA) rules, where the Event specifies when a rule should be triggered, the Condition is a query that is evaluated when the Event occurs, and the Action is executed when the Event occurs and the Condition is satisfied. Events can be seen as signals that inform that a change to data in the database has occurred, e.g. an update of a table. In HiPAC *coupling modes* (fig. 3.1) were defined which specify how the evaluation of rule conditions and the execution of rule actions are related to the detected events and the transaction in which the events occur.

Immediate rule processing means that the rule conditions are evaluated and the actions are executed immediately after the event occurred. A distinction was also made between whether rule processing takes place before or after the change has taken place in the database. *Deferred* rule processing means that rule processing is delayed until the transaction is to be committed. *Casually*

Dependent Decoupled rule processing means that any triggered action execution is executed in a separate sub-transaction that waits until the main transaction is committed. *Decoupled* rule processing means that the sub-transaction is completely decoupled from the main transaction and commits regardless of the outcome of the main transaction.

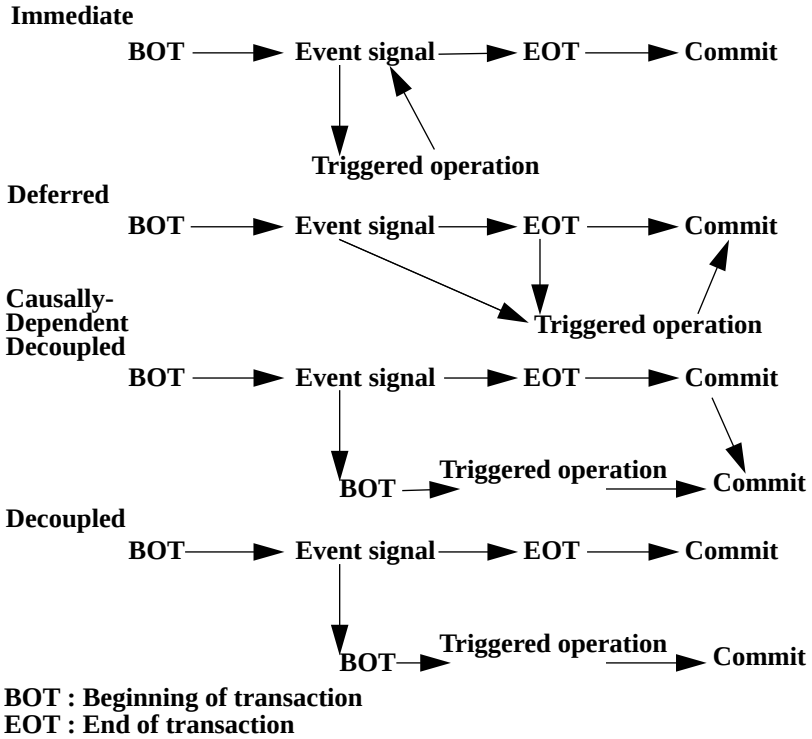


Figure 3.1: Rule processing coupling modes in HiPAC

In POSTGRES [116][133] rules are introduced as ECA rules where events can be *retrieve*, *replace*, *delete*, *append*, *new* (i.e replace or append), and *old* (i.e. delete or replace) of an object (a relation name or a relation column). The condition can be any POSTQUEL query and the action can be any sequence of POSTQUEL commands. Two types of rule systems exist, the *Tuple Level Rule System* which is called when individual tuples are updated, and the *Query Rewrite System* which resides in the parser and the query optimizer. The Query Rewrite System converts a user command to an alternative form, i.e. by wrapping extra code which checks the rules more efficiently. No support exists for handling temporal, external events, or composite events.

In Starburst [84][133] ECA rules are supported which can monitor the events INSERT, DELETE, and UPDATE of a table. The condition can be any

SQL query and the action any sequence of database commands. Rules that are defined can be temporarily deactivated and then be re-activated. The condition and action parts may refer to *transition tables* that contain the changes to a rule's table made since the beginning of the transaction or the last time that a rule was processed (whichever happened most recently). The transition table INSERTED/DELETED contains records inserted/deleted into/from the trigger table. Transition tables NEW_UPDATED and OLD_UPDATED contain new and old values of updated rows, respectively. In [132] the *set-oriented* semantics of Starburst rules are presented. In set-oriented rule execution the action part of a rule is executed for all tuples for which the condition is true in contrast to *instance-oriented* rule execution where it is executed for one tuple at a time.

In Ariel [63][133] production rules are defined on top of POSTGRES. In Ariel CA-rules are allowed which use only the condition to specify *logical events* which trigger rules. Logical events can be expressed by a query or a relational view and specify the logical conditions that are the result of one or several *physical events* (such as an update of a table).

In Ode [55][133] constraints and triggers are introduced into an OODBMS. The *primitive events* that can be referenced are creation, deletion, update, or access by an object method. Ode also supports *composite events* through event expressions that relate primitive events. The event expressions can define sequence orderings between events. A third type of event has been defined, known as an external event, which signals the occurrence of an event outside the database (either in application programs, in the operating system, or in hardware). Other systems based on ECA-rules and which can trigger on external events include REACH [18] and SAMOS [53]. In AMOS external events are introduced as foreign events together with foreign data sources (se chapter 9).

In Chimera [133] different models for processing events, *event consumption modes*, can be specified. Chimera also includes a *debugging mode* where the state of an executing rule can be monitored interactively.

Considerable research has been carried out in the area of active database systems. A good introduction to the research area and active database architectures can be found in [133] which includes overviews of most of the research systems mentioned above (an additional system, A-RDL, is discussed in section 6.3 in this thesis).

3.2 ADBMS Classifications

In the *Active Database Management System Manifesto* [35] required and optional functionality for an ADBMS are presented. Required functionality includes support for creating, modifying, activating, and deactivating (called enabling and disabling) ECA-rules. The ADBMS must support event monitoring and storing events in an *event history* as (<*event type*>, <*time*>) where the <*event type*> represents any primitive event and the <*time*> is the time (transaction time) when the event occurred. The ADBMS must have clearly defined

rule semantics such as the *event consumption* policy (i.e. when events are discarded), event detection (i.e. when events are detected and signalled to the rule manager), and rule semantics such as coupling modes and instance or set-oriented semantics. Some possible event consumption policies are: *recent*, *chronicle*, and *cumulative*. In the recent policy the latest instance of a primitive event that is part of a complex event is consumed if the complex event occurs. In the chronicle policy the events are consumed in time order. In the cumulative policy all instances of a primitive event are consumed if the complex event occurs.

Two new coupling modes are suggested in [35] (fig. 3.2), *Sequential Causally Dependent* (sometimes called Detached) rule processing, where the triggered transaction starts after the triggering transaction is committed, and *Exclusive Causally Dependent* rule processing, where the triggered transaction may commit only if the triggering transaction has failed. *Conflict resolution* policies must also be defined for managing simultaneously triggered rules, e.g. by allowing the user to specify different priorities for conflicting rules. Access to events in condition and action parts of rules might also be defined. Optional functionality includes a rule programming environment with tools such as rule editors, rule browsers, rule analyzers, rule debuggers, trace facilities, and performance tuning tools. Some examples of tools in a support environment for

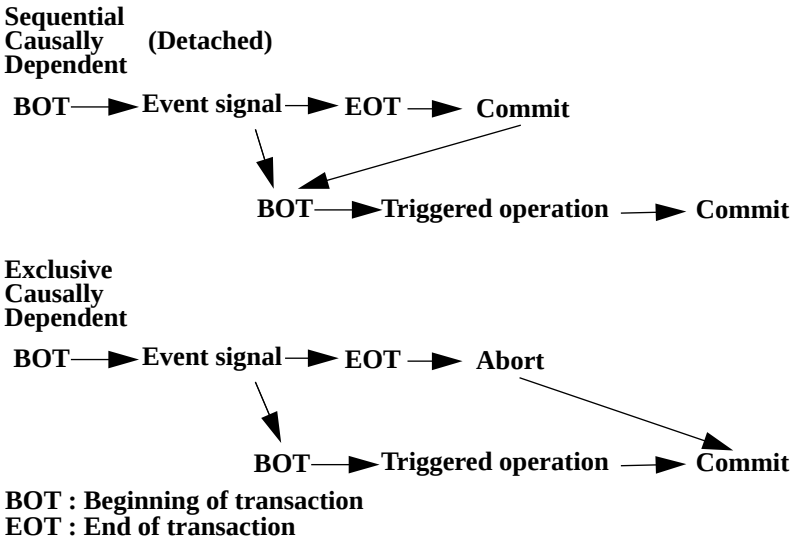


Figure 3.2: Two newly suggested rule processing coupling modes

active rule design can be found in [8].

Two important aspects for comparing different architectures are the expressiveness of the rule language and the execution semantics of the rules.

The expressiveness of the rules can be divided into the expressiveness of rule events, conditions and actions. The expressiveness of the event part can be

divided into comparing the types of events that the rules can reference and how the events can be modelled and combined into complex events. Different types of events include database updates, schema changes, and external events such as sensor value changes, specified state changes in the applications, or time. Modelling events can include an event specification language that can combine events using logical composition, event ordering, sequential and temporal ordering, and event periodicity [24].

The expressiveness of the condition part can be divided into whether a full query language is available or not, whether the events can be referenced as changed data, and whether old values can be referenced or not.

The expressiveness of the action part can be divided into whether a full query language is available or not, i.e. whether queries and updates can be intertwined, and whether the action can include schema changes and rule activation/deactivation.

Execution semantics of rules includes rule processing coupling modes defined in section 3.1. If full query language expressiveness is possible in the condition part, then set-oriented rule semantics are also possible [132], where the action part is executed over a set of tuples produced by the condition. Cascading rule execution, i.e. whether one rule can trigger another, and if simultaneously triggered rules are subjected to some conflict resolution method are also part of the classification of rule semantics.

In [133] different architectures of ADBMSs are defined as *layered architectures*, *built-in architectures*, and *compiled architectures*.

- In a layered architecture the rule system is loosely coupled with the DBMS by intercepting client-server communication and by calling application procedures or submitting commands to the DBMS. Layered architectures are usually easier to implement, but can exhibit poor performance.
- In a built-in architecture the rule system is tightly coupled with the DBMS and rule processing is integrated with query processing. Built-in architectures usually provide good performance, but are substantially more difficult to implement.
- In a compiled architecture the rule system is an extension of the application or database query language where the rules are wrapped as procedural code around expressions that generate events that might affect the rule. A compiled architecture requires that the compiler can detect all events at compile-time which is usually not the case in general database interfaces where applications are allowed to perform ad-hoc modifications to the database.

In [35] ADBMSs are classified according to how they are used by applications, i.e. for monitoring or for control. A classification is also made according to how the applications are integrated with the ADBMS. The ADBMS is often considered as a stand-alone system with applications as clients that connect to the ADBMS server. Alternatively, the ADBMS can be integrated (embedded) in a system as a component that can be used by applications, but where the applications are considered as providing the main functionality of the whole system.

The AMOS ADBMS is based on a built-in architecture where rule process-

ing is tightly integrated with query processing. Active rules in AMOS are primarily designed for efficient monitoring of changes to the database, but can be used for control as well. AMOS can be run as a stand-alone system or be tightly integrated with applications.

3.3 AMOS

AMOS [41] (Active Mediators Object System) is a system that can model, locate, search, combine, and monitor data in information systems with many workstations connected using fast communication networks. The architecture uses the *mediator* approach [131] that introduces an intermediate level of software between databases and their use in applications and by users. We call our class of intermediate modules *active mediators*, since our mediators support active database facilities. The AMOS architecture is built around a main memory-based platform for intercommunicating information bases. Each AMOS server has DBMS facilities, such as a local database, a data dictionary, a query processor, transaction processing, and remote access to databases. The AMOS multi-DBMS architecture is presented in Paper III, [42], and [130]. The AMOS DBMS is an extension of a main-memory version of Iris [47], called WS-Iris [82], where OSQL queries are compiled into execution plans in an OO logical language, ObjectLog [82]. The query language of AMOS, AMOSQL, is a derivative of OSQL [85]. AMOSQL extends OSQL with active rules, a richer type system, and multi-database functionality.

In fig. 3.3 the AMOS architecture (excluding the multi-DBMS parts) can be seen where the different levels and modules are:

- The *external application interface* level can handle embedded AMOSQL by sending the expressions to the level below for parsing and execution. An AMOS fast-path interface that does not require any parsing is also available. Results are returned to the external interface, either directly or through interface variables and cursors.
- The *AMOSQL interpreter* parses AMOSQL expressions and sends requests to the levels below. AMOSQL supports precompiled functions (views and stored procedures) and most applications will only parse and optimize complicated queries once and then call the functions directly.
- 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 rule compilation, activation/deactivation, monitoring of events, and execution of rules and is described in more detail in this chapter.
- The *event manager* dispatches events received on the *event bus* to the rule processor. Events can come from *internal events* intercepted by the transaction manager such as schema updates or relational updates. Other possible events are *foreign events* from foreign data sources and *time events* from the agenda. The event manager also supports storing events in event histories represented as time

series that can be accessed through event functions. The event functions can be accessed by the rule processor through AMOSQL queries.

- The *agenda* is a time management module that can schedule activities to be performed at specific times. The agenda is more discussed in section 9.3.6.
- The *foreign data source interface* supports extension of AMOS with new data structures and interfaces to other non-local data. It interacts with the AMOSQL optimizer since access of new data structures such as available indexes are crucial for query optimization. It interfaces with the ObjectLog interpreter since compiled execution plans will need access to foreign data. It interfaces with the logi-

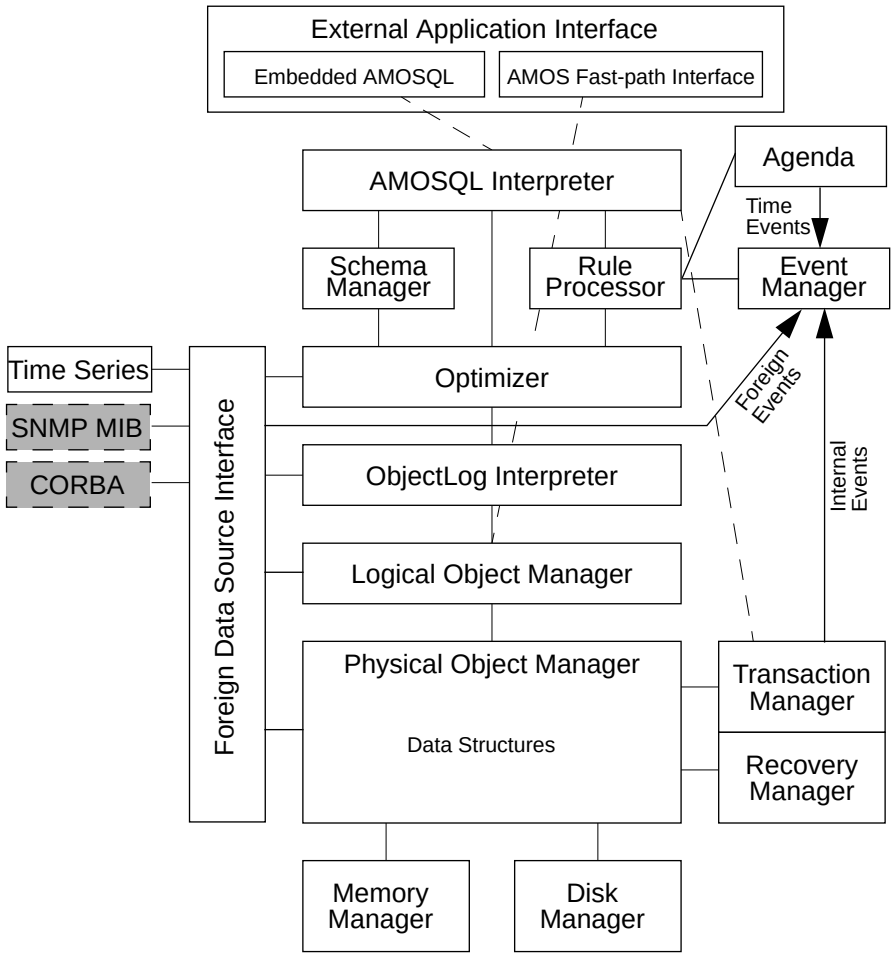


Figure 3.3: The AMOS architecture

cal object manager since many foreign data sources need to create special objects, e.g. time series are used for storing the data of event functions and interfaces such as CORBA and SNMP MIB (see section 9.5.3) would need to create interface objects and *foreign functions* that are accessible through AMOSQL queries. Finally it also needs to interface with the physical object manager since new data structures such as time series have to be defined together with operations for allocation, deallocation, access, and updating. Foreign data sources in general are discussed in chapter 9.

- The *AMOSQL optimizer* is responsible for transforming ad hoc queries, update statements, functions, and stored procedures into efficient execution plans using query optimization and compilation techniques. This process involves the application of transformation rules and heuristic cost-based query optimization techniques that produce executable and efficient query plans. By supporting the definition of execution costs for foreign functions (default costs are also provided), the optimizer can optimize expressions that include foreign functions as well. Query optimization in AMOS and the management of foreign predicates are presented in [82] and [48].
- The *ObjectLog interpreter* [82] supports efficient execution of optimized query plans. All data is accessed here through logical predicates. Stored functions, i.e. tables, are represented as facts and derived functions, i.e. views, are represented as Horn Clauses. Foreign functions are represented as foreign predicates that can access the foreign data through the foreign data source interface.
- The *logical object manager* manages all operations to all objects in the database schema such as object creation, deletion, and updates of object attributes including updating, inserting, and deleting data in stored functions, i.e. base relations. This level also handles OIDs (Object Identifiers) of the logical objects. All operations on these objects are transactional and are thus logged. All operations generate events that are intercepted and sent to the event manager.
- The *physical object manager* handles all basic objects (everything in the database is an object) such as atoms, strings, integers, real numbers, lists, arrays, hash tables, tree structures, and time series.
- The *transaction manager* handles all database transactions by keeping an undo/redo log of all database operations. It also intercepts logged operations such as updates and schema changes and passes them as events to the event manager through the event bus.
- The *recovery manager* ensures persistency by making periodical snapshots and flushing the log to disk.
- The *memory manager* manages all memory operations such as allocation, deallocation, and garbage collection.
- The *disk manager* in AMOS is more primitive in comparison to disk-based DBMSs. It mainly handles flushing of database images and logs between main-memory and disk for initiation, connection, or saving of databases.

The event handling is tightly integrated into the system and internal changes are intercepted where they occur in the lower levels for efficiency reasons. The rule processor is tightly integrated with the query processing for the same reason.

3.4 The Rule Processor and the Event Manager

The active rules in AMOS are of Condition Action (CA), Event Condition Action (ECA), and Event Action (EA) types. The AMOS rule processor handles rule creation/deletion, activation/deactivation, monitoring, and execution. The processing of rules is divided into four phases:

1. Event Detection
2. Change monitoring
3. Conflict resolution[1]
4. Action execution

Event detection consists of detecting events that can affect any activated rules and is performed continuously during ongoing transactions. Events are accumulated in event histories represented by *event functions*. Complex event detection (in ECA-rules) is performed by executing logical expressions (basically simple queries) over several event functions. Change monitoring includes using the *event data* from the event functions to determine whether any condition of any activated rules have changed, i.e. have become true. During action execution further events might be generated causing all the phases to be repeated until no more events are detected. Different conflict resolution methods are outside the scope of the thesis. In the current implementation a simple priority based conflict resolution is used.

The rule execution model in AMOS is based on the *Event Condition Action* (ECA) execution cycle (fig. 3.4). All events are sent on an *event bus* that queues the events until they are processed. The execution cycle is always initiated by non-rule-initiated events such as database updates, schema changes, time events, or other external events. In AMOS events are intercepted in a similar manner as in POSTGRES [116] and Starburst [84]. However, the events that are intercepted in AMOS include all operations of logical objects. This makes it possible to extend rules to trigger on any change in the system, including schema updates. All events are dispatched through table-driven execution. Events are accumulated chronologically in stored temporal event functions represented by time series. More about the temporal aspects of event functions can be found in chapter 8. Event functions are used in AMOSQL *event expressions* that can define complex events. The event expressions are automatically generated from analyzing rule conditions for CA-rules and are created from the event part of ECA and EA-rules. The event expressions generate event data which is

1. Conflict resolution is the process of choosing one single rule when more than one rule is triggered.

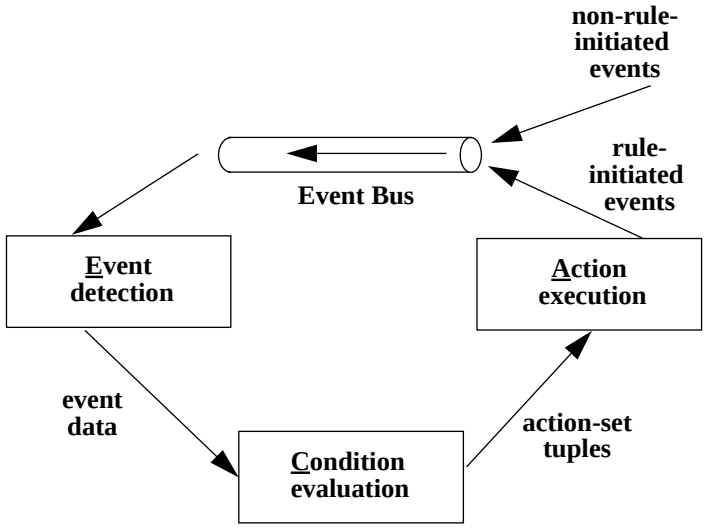


Figure 3.4: The ECA execution cycle

used for evaluating the rule conditions.

Rule checking is performed in a *check phase* usually at transaction commit (deferred rule checking). Rule checking can also be invoked by calling the *check procedure* explicitly. During rule checking, rule conditions are evaluated if there is any event data, since this signals that the rule has been triggered. The evaluation of the rule conditions produces *action-sets* that contain tuples for which the actions are to be executed. If an action-set is empty, this signifies that the rule condition was false. When the actions are executed, new events might be generated and the execution cycle continues until no more events are detected on the event bus.

The active rules in AMOS can be classified according to the features presented in section 3.2. The expressiveness of events is based on logical composition (AND, OR), event ordering (BEFORE, AFTER), and simple temporal events (AT <time point>, WITHIN <time interval>). Events can be *primitive*, internal (such as ADD, REMOVE, and UPDATE of tables and CREATE/ DELETE of objects) and external (such changes to a sensor). Events can also be *complex* such as ADD, REMOVE, and UPDATE of relational views. The expressiveness of conditions is based on the availability of complete AMOSQL queries in the condition. The expressiveness of actions is based on full AMOSQL procedural statements, i.e. queries intertwined with any updates of the schema, updates of functions, rule activation/deactivation, and application call-backs. The rules in AMOS have as default a deferred coupling mode, but other coupling modes, such as immediate, sequential causally dependent (or detached), and manual invocation of rule checking, can be used by using different rule contexts (see Paper IV and section 7.8).

3.5 The Iris Data Model and OSQL

The data model of AMOS and AMOSQL is based on the data model of Iris [47] and OSQL [85]. The Iris data model is based on objects, types, and functions (fig. 3.5). Everything in the data model is an object, including types and functions. All objects are classified by belonging to one or several types, which equal object classes. Types themselves are of the type 'type' and functions are of the type 'function'.

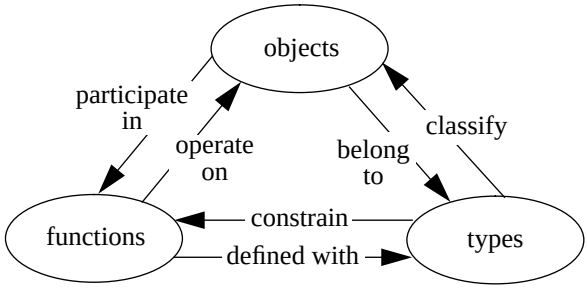


Figure 3.5: The Iris data model

The data model in Iris is accessed and manipulated through OSQL[1]. (All examples of actual schema definitions and database queries will here be written in a courier font.) For example, it is possible to define user types and subtypes:

```
create type person;
create type student subtype of person;
create type teacher subtype of person;
create type course;
```

Stored functions can be defined on types that equal attributes in Object-Oriented databases or base relations in Relational databases; hence we call this model *Object Relational* [118]. One function in the Iris data model equals several functions in a mathematical sense. For example, the built-in `sqr` function can be used for calculating both the square of a number and the square root. By calling `sqr` in a query with the argument unbound and the result bound to some number, the AMOSQL compiler will choose an internal function that calculates and binds the argument of `sqr` to the square root of the result.

Let us define a function that can both give the name of a person given the person object or give all the person objects associated with a name.

```
create function name(person) -> charstring as stored;
```

Stored functions is the default so the '`as stored`' part can be omitted:

1. Some syntax here, especially for stored procedures, is actually AMOSQL.

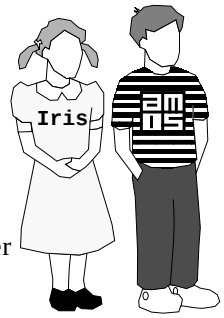
```
create function studies(student) -> bag of course;
create function passed(course) -> bag of student;
create function gives(teacher) -> bag of course;
```

Bags are used for storing *multi-valued* functions. Derived functions equal methods or relational views and can be defined in terms of stored functions (and other derived functions).

```
create function teaches(teacher t) -> student s
       as select s for each course c where
       gives(t) = c and
       c = studies(s);
```

Note that derived functions such as the `teaches` function implicitly return bags, even though they are declared as returning a single type, since they are the result of a query. Queries return a stream of data that can accessed one at a time or be collected into a bag.

Instance objects of a type can be created and stored functions can be set for these instances:




```
create student instances :iris[1], :amos;
set name(:iris) = “Iris”;
set name(:amos) = “AMOS”;
create course instances :active_DBMSs;
```

All user-defined objects will be given an Object Identifier (OID). Single values can be added to (and removed from) multiple-valued functions.

```
add studies(:amos) = :active_DBMSs;
remove studies(:amos) = :active_DBMSs;
```

Functions can be defined with *multiple arguments and values*, i.e. with tuple results.

```
create function grade(student) ->
                            bag of <course, charstring>;
```

Stored procedures are defined as functions that have side-effects:

```
create function teach(teacher t, student s, course c)
       -> boolean as
       begin
         if (s = passed(c))
         /* the student has already passed the course */
```

1. These are interface variables and are not part of the database.

```
         then result false
         else
         begin
            /* if teacher t is not already teaching the
               course c, mark t as teacher of course c */
            if notany(select gives(t) = c)[1]
            then add gives(t) = c;
            /* if student s is not already taking the
               course c, add s as student of course c */
            if notany(select studies(s) = c)
            then add studies(s) = c;
            result true;
         end;
      end;

create function mark(student s, course c, charstring g)
      -> boolean as
      begin
         if (c = studies(s))
         /* the student is taking the course */
         then
         begin
            add grade(s) = <c, g>;
            if g != "Failed" then add passed(c) = s;
            remove studies(s) = c;
            result true;
         end
         else
         /* the student is not taking the course */
         result false;
      end;
```

As can be seen in the examples above, stored procedures can access functions and perform ad hoc queries. Stored procedures are, however, not allowed in queries (and derived functions) since a series of queries should always return the same result regardless of in what order the queries are executed[2], i.e. they cannot contain side-effects.

Multiple inheritance, i.e. multiple supertypes, is possible by creating a type with two supertypes:

```
create type student_teacher subtype of student, teacher;
```

1. Bags can be created at run-time by sub-select expressions, and `notany` is an aggregate function that returns true if it is called with an empty bag.
2. This is crucial for query optimization since the query optimizer will reorder queries to generate an efficient execution plan.

New types[1] can be added to an instances:

```
add type student_teacher to :amos;
```

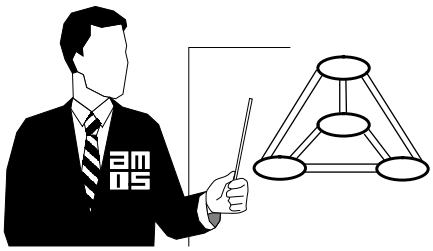
Procedures are called by:

```
call teach(:amos, :iris, :active_DBMSs);
TRUE
```

Functions are *multi-directional* which means that they can be accessed in *inverse queries* where the result value is known and the argument(s) is(are) required.

```
select name(t)
  for each teacher t
  where teaches(t) = :iris;

"AMOS"
```



In the previous example the last query returns a single tuple. Queries, and subsequently functions, can return several tuples. Duplicate tuples are removed from stored functions if they are not explicitly defined to return a bag. We say that we have *set-oriented semantics*. *Bag-oriented semantics* is available as an option and can be specified along with the return type of a function as defined for the `studies` function.

Functions can be overloaded on the types of their arguments, i.e. we can define the same function in several ways depending on the types of the arguments. The system will in most cases choose the correct function at compile time, this is known as *early binding*. In some cases the system cannot determine what function to choose at compile time and the execution plan must check some types at run time, this is known as *late binding*. Let us define a new function `name` that overloads on the first argument:

```
create function name(course) -> charstring;
```

The AMOSQL compiler will choose the correct versions of the `name` function by looking at the type of the arguments (early binding). If, however, the exact type of the argument is unknown, e.g. the type is only specified as `object` (the

1. Instances belong to a set of types (usually the immediate supertype with its supertypes) and a new added type must be a subtype of one of the current types of the instance.

most general type) at compile-time, but is known at run-time, then the choice will be made at run-time (late binding). If for some reason the exact type is not known at run-time either, it is possible to help the AMOSQL compiler by specifying the full names of the functions. Full names of functions are specified by the complete function signature: <arg_1 type>. ...<arg_m type>.<function name>`->`<res_1 type>. ...<res_n type>, e.g. `person.name->charstring` and `course.name->charstring`.

Interface variables are untyped if they are not declared[1] so the following expression must be manually type resolved:

```
set course.name->charstring(:active_DBMSs) =
       “Active Database Management Systems”;
```

On the top-level `select` and `call` can be left out:

```
course.name->charstring(:active_DBMSs);
“Active Database Management Systems”
mark(:iris, :active_DBMSs, “Excellent”);
TRUE
```

AMOS also allows the introduction of functions written in some other programming language such as C and these are known as *foreign functions*:

```
create function print(charstring) -> boolean as
foreign “Cprintfn”[2];
create function print_grades(course c) -> boolean as
begin /* prints course grades on a console or printer */
       print(name(c));
       for each student s, charstring g
       where grade(s) = <c, g>
          print(name(s) + ”: “ + g);[3]
end;

print_grades(:active_DBMSs);
Active Database Management Systems
Iris: Excellent
```

Different access patterns and cost information can be specified for foreign functions to support inverse queries and query optimization. See [82] for more details.

1. Variables can be declared by: `declare course :active_DBMSs;`
2. Bindings to foreign functions are resolved during linking and at system initialization.
3. The ‘+’ operator is overloaded on `charstring` with a foreign function for string concatenation.

A transaction can be completed and made permanent by:

```
commit;
```

A transaction is aborted and rolled back by:

```
rollback;
```

More information about the AMOSQL compiler and optimizer can be found in [48][82]. Since types and functions are objects as well, of the types ‘type’ and ‘function’, it is possible to define generic functions, i.e. functions that take types as arguments, and higher order functions, i.e. functions that take other functions as argument.

3.6 About Paper II

This paper was the first publication on adding rules to AMOS. The rules presented in this paper have later been implemented. Note that the rule syntax in Paper II differs slightly from the syntax in this chapter which is based on the actual implementation. To distinguish this new extended query language from that of Iris we decided to change the name to AMOSQL.

3.7 The AMOS Data Model and AMOSQL

The AMOS data model extends that of Iris by introducing rules (fig. 3.6). Rules are also objects [30] and of the type ‘rule’. AMOS is also based on the *functional data model* of Daplex [108] and the active rules of AMOS are based on a functional model. Rules monitor changes to functions and changes to functions can trigger rules. All the events that the rules can trigger on are modelled as changes to values of functions. This gives us the power of AMOSQL functional expressions as our event modelling language. Functions are seen as having passive (synchronous) or active (asynchronous) behaviour depending on whether they are used in a query or in a rule condition. Passive functions display synchronous polling behaviour, i.e. query answering behaviour, while active functions display asynchronous interrupt behaviour, i.e. event signalling behaviour. This is similar to the idea of *fluents* [104] as functions for modelling dynamic behaviour. Purely passive functions are functions that never change their extent, such as built-in arithmetic functions, e.g. `+`, `-`, `*`, and `/`, boolean functions, e.g. `=`, `<` and `>`, and aggregate functions such as `sum` and `count`. Functions that are defined in terms of these functions can have their values changed, but never the purely passive functions themselves.[1]Foreign functions written in some procedural language were initially also considered to be purely passive functions, but this was later changed with the introduction of *foreign data*

1. It would be strange to monitor changes to `plus`, e.g. if `1+1` were to become `3`.

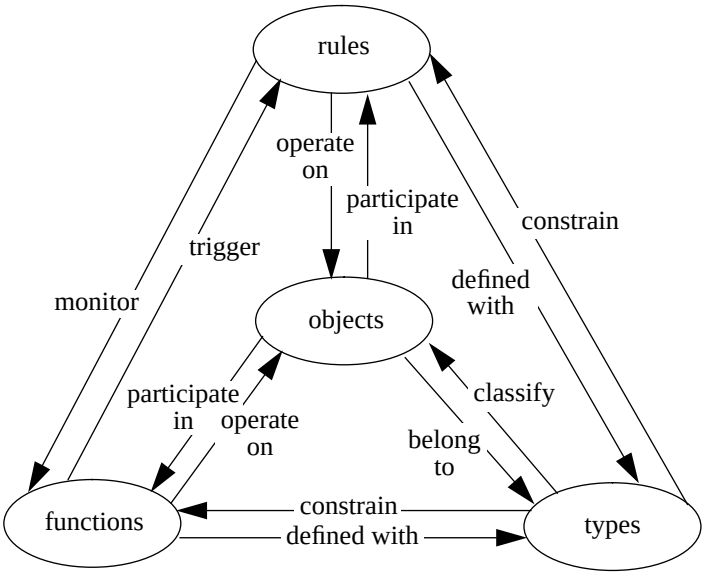


Figure 3.6: The AMOS data model

sources (see chapter 9).

The first version of the rule system (using CA-rules) did not have any purely active functions, but these would have been *event functions*, i.e. functions that represent internal or foreign events. In some cases it is desirable to directly refer to specific events such as added or removed; these can be modelled as specific event functions that change if tuples are added/removed to/from a specific function. This was later implemented in a rule system supporting ECA-rules (as well as CA and EA-rules) where changes to stored functions are defined through three event functions for added, removed, and updated tuples. Event functions that represent external changes are active foreign functions or foreign data sources and can be functions representing sensors in a CIM application or the status of network elements in an ATM network (see chapter 9).

The CA-rules presented here have conditions that reference stored and derived functions only. The events that trigger these conditions are the function update events, or events from adding or removing tuples to/from functions. Stored and derived functions can be seen as having active behaviour if they are referenced in event expressions in ECA and EA-rules, or in event expressions derived from CA-rules. Functions can be seen as having passive behaviour if referenced inside queries. Only functions without side-effects, i.e. no stored procedures, are allowed in rule conditions.

The rule processor calculates all the events that can affect a CA-rule condition. This is the default for rule condition specifications and can be seen as a *safe* way to avoid users forgetting to specify relevant events, as can happen with traditional ECA-rules. ECA-rules are sometimes needed if the user, for

some reason, wants the rule to disregard some events that would be automatically monitored by a CA-rule (such as updates that are allowed to violate the condition of an integrity rule) or if the user wants to monitor some event that would be ignored by a CA-rule (such as specific temporal ordering of events).

By modelling rules as objects it is possible to make queries over rules. Overloaded and generic rules are also allowed, i.e. rules that are parameterized and can be activated for different types. However, unknown type information during rule compilation will cause late binding, i.e. run-time type checking, and will degrade rule processing performance.

In AMOSQL, OSQL is extended with rules having a syntax conforming to that of OSQL functions. AMOSQL supports rules of CA type where the condition is an AMOSQL query, and the action is any AMOSQL procedure statement, except `commit`. Data can be passed from the condition to the action of each rule by using shared query variables, i.e. set-oriented action execution [132] is supported.
The syntax for creating and deleting CA-rules is as follows[1]:

create rule *rule-name parameter-specification* **as**
    [*for-each-clause*]
    when *predicate-expression*
    do *procedure-expression*
where
for-each-clause ::=
    for each *variable-declaration-commalist*

delete rule *rule-name*

The *predicate-expression* can contain any boolean expression, including conjunction, disjunction, and negation. Rules are activated and deactivated by:

activate rule *rule-name* ([*parameter-value-commalist*]) [**priority** 0|1|2|3|4|5]
deactivate rule *rule-name* ([*parameter-value-commalist*])

Rules can be activated/deactivated for different argument patterns. The semantics of a rule 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 rule activation. If something happens later in the transaction which causes the condition to become false again, the rule is no longer triggered. This ensures that we only react to *logical events*. The truth value of a condition is here represented by *true* for a non-empty result of the query that represents the condition and *false* for an empty answer.

When the condition of a triggered rule activation is evaluated, it is executed separately with its actual parameter values. After the evaluation of the condition the values of any shared variables between the condition and action are

1. Note that the syntax differs slightly from that in the published papers.

saved in an *action-set* for each rule activation.

In the current implementation a simple *conflict-resolution* method, based on priorities, is used to specify the order of action execution of rule activations that are simultaneously triggered. Rule activations with corresponding action-sets are stored as *scheduled rule activations* in a priority queue based on the priority of the rule activation. The scheduled rule activations are then fetched in priority order and each action is evaluated using the corresponding action-set. Any duplicates are removed from the action-set to give true set-oriented rule execution.

Some examples of AMOSQL rules are given below.

A classic example for active databases is that of monitoring the quantity of items in an inventory. When the quantity of an item drops below a certain threshold, new items are to be automatically ordered.

```
create type item;
create type supplier;
create function quantity(item) -> integer;
create function max_stock(item) -> integer;
create function min_stock(item) -> integer;
create function consume_frequency(item) -> integer;
create function supplies(supplier) -> item;
create function delivery_time(item, supplier)
            -> integer;
create function threshold(item i) -> integer as
     select consume_frequency(i) * delivery_time(i, s)
                          + min_stock(i)
     for each supplier s where supplies(s) = i;

create rule monitor_item(item i) as
     when quantity(i) < threshold(i)
     do order(i, max_stock(i) - quantity(i));[1]
```

This rule monitors the quantity of an item in stock and orders new items when the quantity drops below the threshold (fig. 3.7) which considers the time to get new items delivered (where `order` is some procedure that does the actual ordering). The consume-frequency defines how many instances of a specific item are consumed on average per day.

For example, the following definitions ensure that the quantity of shoelaces in the inventory is always kept between 100 and 10 000 (if the supplier delivers on time) and will trigger the rule if the quantity drops below 140.

```
create item instances :shoelaces;
set max_stock(:shoelaces) = 10000;
set min_stock(:shoelaces) = 100;
```

1. In AMOSQL `select` and `call` are syntactic sugar and are optional on the top-level.

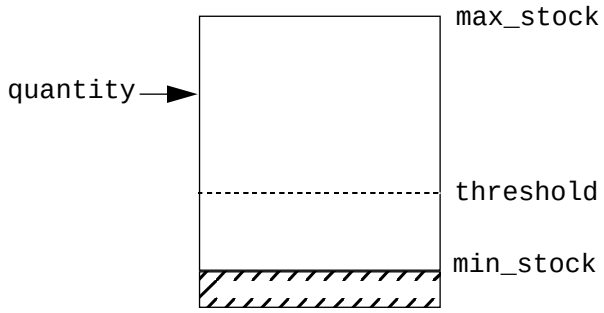


Figure 3.7: Monitoring items in an inventory

```
set consume_frequency(:shoelaces) = 20;
create supplier instances :shoestring_inc;
set supplies(:shoestring_inc) = :shoelaces;
set delivery_time(:shoelaces, :shoestring_inc) = 2;
activate rule monitor_item(:shoelaces);
```

A rule that monitors all items can be defined as:

```
create rule monitor_items() as
     for each item i
     when quantity(i) < threshold(i)
     do order(i, max_stock(i) - quantity(i));
```

In real life there will probably be several suppliers for one item and with different prices. In this case the rules should really consider the minimum threshold, i.e. the supplier who can deliver the fastest and at an acceptable cost.

Another example of rules in active databases is that of *constraints*. If we want to ensure that the `quantity` of an item can never exceed the `max_stock` of that item, we can express this in the following rule:

```
create rule check_quantity() as
     for each item i
     when quantity(i) > max_stock(i)
     do rollback;
```

If this rule is triggered by too many items being ordered, it is not enough to just roll back the transaction. Sometimes *compensating transactions* are needed that undo some external operation such as undoing an order for too many items by returning excess items to the supplier. See section 5.5 for further discussion about compensating transactions.

The previous rules did not really use any of the OO capabilities of AMOSQL, i.e. there was only a flat set of user-defined types. To illustrate OO capabilities, take as an example a rule that ensures that no one at a specific

department has a higher salary than his/her manager. Employees are defined as having a name, an income, and a department. The net income is defined based on 25% tax for both employees and managers, but with a bonus of 100 before tax for managers. Departments are defined as having a name and a manager. The manager of an employee is derived by finding the manager of the department to which the employee is associated. The rule `no_high` is defined to set the income of an employee to that of his/her manager if he/she has a net income greater than his/her manager. The AMOSQL schema is defined by:

```
create type department properties (name[1] charstring);
create type employee properties
       (name charstring, income number, dept department);
create type manager subtype of employee;
create function grossincome(employee e) -> number as
       select income(e);
create function grossincome(manager m) -> number as
       select income(m) + 100;
create function netincome(employee e) -> number as
       select employee.grossincome->number(e) * 0.75;
create function netincome(manager m) -> number as
       select grossincome(m) * 0.75;
create function mgr(department) -> manager;
create function mgr(employee e) -> manager as
       select mgr(dept(e));

create rule no_high(department d) as
    for each employee e
    when dept(e) = d and
         employee.netincome->number(e) >
                                    netincome(mgr(e))
    do set employee.grossincome->number(e) =
                                    grossincome(mgr(e));
```

Note that the functions `grossincome`, `netincome`, and `mgr` are overloaded on the types `employee`, `manager`, and `department`, `employee`. For the function calls `grossincome(m)`, `grossincome(mgr(e))`, `netincome(mgr(e))`, `mgr(dept(e))`, and `mgr(e)` this is resolved at compile time; we call this *early binding*. This is possible since the actual parameters in the calls are of distinct types. In cases when the compiler cannot deduce what function to choose, the complete function signature, e.g. `employee.netincome>number(e)`, can be specified to aid the compiler to choose the correct function at compile time. In the rule condition, `employee.netincome->number` can be called for all employees, including managers, since managers are employees as well. If `e` is a manager, the rule will check if the manager makes more than his/her manager; if there is no manager above him/her, the condition will be considered false since the answer to the query `mgr(e)`

1. This is shorthand for defining a stored function, `name`, on departments.

will be empty.

In cases when the compiler cannot deduce what function to choose, it will produce a query plan that does run-time type checking to choose the correct function; we call this *late binding*. Look at the following redefinition of the `no_high` rule:

```
create rule no_high(department d) as
    for each employee e
    when dept(e) = d and
         netincome(e) > netincome(mgr(e))
    do set employee.grossincome->number(e) =
                                    grossincome(mgr(e));
```

Different `netincome` functions will here be chosen depending on whether the argument it is called with is just an employee, or a manager as well. The rule condition is different than that of the previous rule since, if the `employee e` is a manager, the net income will be calculated differently. This is because `manager.netincome->number` would, in this case, be chosen in both instances in the condition. This rule is more elegant, but in order not to complicate the generated code and the discussion of change-monitoring techniques in the following chapters, the first version of `no_high` will be used in the continuation of the example.

Note that the `employee.grossincome->number` function is updatable since it is directly mapped to the stored function `employee.income->number`. The function `manager.grossincome->number` is, however, not directly updatable since it cannot be directly mapped to a stored function. This is described in more detail in [82].

The `no_high` rule will be activated for a specific department and will serve as an example in the rest of the section.

Let us define a toys department with a manager and five employees:

```
create department(name) instances
       :toys_department("Toys")[1];
create manager(name, dept, income) instances
       :boss("boss", :toys_department, 10400);
set mgr(:toys_department) = :boss;
create employee(name,dept,income) instances
       :e1("employee1",:toys_department,10100),
       :e2("employee2",:toys_department,10200),
       :e3("employee3",:toys_department,10300),
       :e4("employee4",:toys_department,10400),
       :e5("employee5",:toys_department,10500);
```

The employees with their incomes and netincomes can be seen in fig. 3.8.

Now, if we activate the rule for the toys department and try to commit the transaction, a check is made as to whether any of the employees have a net

1. This is a short-hand for setting the function `name`, for a department instance.

income higher than their manager. No such employees exist and thus the rule is not triggered.

.

name	income	netincome
boss	10400	7875
employee1	10100	7575
employee2	10200	7650
employee3	10300	7725
employee4	10400	7800
employee5	10500	7875

Figure 3.8: Initial employee salaries

```
activate rule no_high(:toys_department);
commit; /* check and commit */
```

Now if we change the income of employee2 and employee4:

```
set income(:e2) = 10600;
set income(:e4) = 10600;
```

we can see in fig. 3.9 that the netincomes of employee2 and employee4 exceed that of their manager.

name	income	netincome
boss	10400	7875
employee1	10100	7575
employee2	10600	7950
employee3	10300	7725
employee4	10600	7950
employee5	10500	7875

Figure 3.9: Employee salaries before commit

If we try to commit this transaction, the `no_high` rule will be triggered and the salaries of employee2 and employee4 will be set to that of their manager. This can be seen in fig. 3.10.

```
commit; /* check and commit */
```

name	income	netincome
boss	10400	7875
employee1	10100	7575
employee2	10500	7875
employee3	10300	7725
employee4	10500	7875
employee5	10500	7875

Figure 3.10: Employee salaries after commit

In this example, the rule condition monitoring consists of determining changes to the condition of the `no_high` rule. Changes to several stored functions (i.e. `dept`, `income`, and `mgr`) can affect the rule condition. In the example, only two updates are made to the `income` function. The rule-condition monitoring must be efficient even if the number of employees is very large. However, evaluating the condition of `no_high` naively (i.e. evaluating the whole query of the rule condition) would result in checking the income of all employees for the department. Efficient techniques for evaluating rule conditions based on changes that result from small updates, such as in these previous examples, are discussed in chapter 6.

Note that the rules can also be invoked explicitly at any time during a transaction by calling the check procedure:

```
check();
```

Rules can also be grouped into rule contexts that can be passed as argument to the check procedure (see Paper IV and section 7.8).

3.8 ECA-rules

CA-rules do not always provide all the control over the rules and their behav-

iour that is sometimes needed. A CA-rule can be translated to an ECA-rule where the event part is a disjunction of all the events that can affect the condition. In some situations it is desirable to separate this rule into several ECA-rules that perform different actions depending on which event triggered the rules. An ECA-rule can be written to disregard events that a CA-rule would monitor and to monitor events that a CA-rule would ignore.

3.9 ECA-rules in AMOS

The implementation of the rule system in AMOS has continued with the introduction of explicit events and ECA-rules [86]. The syntax for the rules has now changed to:

create rule *rule-name parameter-specification*
[*for-each-clause*]
[**on** *event-type-specification*]
[**when** *predicate-expression*]
do *procedure-expression*

delete rule *rule-name*

where

for-each-clause ::=
for each *variable-declaration-commalist*

This allows for writing ECA-rules, CA-rules, and EA-rules. The main difference from the CA-rule syntax is the explicit event specification. The event, the condition, and the action part can all share the same variables to allow data to be passed between the parts of the rule during rule execution. The events that can be specified include:

event-type-specification ::=
added(*function-call*) |
removed(*function-call*) |
updated(*function-call*) |
created(*variable-name*) |
deleted(*variable-name*)[1] |
foreign-event-name |
event-type-specification **and** *event-type-specification* |
event-type-specification **or** *event-type-specification* |
event-type-specification **before** *event-type-specification* |
event-type-specification **after** *event-type-specification*

1. Actually not yet implemented due to technical problems in AMOS on how to reference objects that have been marked as deleted. Here the marking of deleted objects can be delayed or an immediate coupling mode with checking before operations take effect is needed.

The added, removed, and updated event types monitor changes to stored and derived functions. The created and deleted event types monitor the creation and deletion of object instances of some certain object type.

The semantics for ECA and EA-rules is: If enough events occur to make the event specification of an activated rule true, then the rule is marked as triggered for this rule activation. Any event data, i.e. shared variables between event and condition, action parts, is saved. If the rule has a condition, it is evaluated with the event data. The data shared between the action and both the event and the condition is then saved as the action-set for each rule activation. Any duplicates in event data are removed to avoid multiple triggering on the same events. Any duplicates are removed from the action-set to give true set-oriented rule execution. Action execution is then scheduled as was defined for CA-rules.

Take the rule:

```
create rule eca_no_high(department d) as
    for each employee e, manager m
    on updated(income(e)) or updated(income(m)) or
       updated(dept(e)) or updated(mgr(e))
    when dept(e) = d and
         m = mgr(e) and
         employee.netincome->number(e) > netincome(m)
    do set employee.grossincome->number(e) =
                                    grossincome(m);
```

This rule has identical behaviour to the CA-rule `no_high`. It is, however, more cumbersome to write. When a CA-rule is compiled, the rule compiler will automatically deduce all involved events and assume an implicit disjunction between them. If we wanted to define a rule that only triggers when an employee gets a salary raise and ignore all other events (such as if an employee changes department or manager), we can write:

```
create rule no_raise(department d) as
    for each employee e, manager m
    on updated(income(e))
    when dept(e) = d and
         m = mgr(e) and
         employee.netincome->number(e) > netincome(m)
    do set employee.grossincome->number(e) =
                                    grossincome(m);
```

This rule is impossible to write as a CA-rule and shows the need for having ECA-rules as well. Most ADBMSs with ECA-rules only support specifying events relating to tables (stored functions in AMOSQL) and not events relating to views (derived functions in AMOSQL). In AMOS it is possible to specify events relating to derived functions as well [86]. The `no_raise` rule could be written as:

```
create rule no_raise(department d) as
    for each employee e, manager m
    on updated(employee.netincome->number(e))
    when dept(e) = d and
         m = mgr(e) and
         employee.netincome->number(e) > netincome(m)
    do set employee.grossincome->number(e) =
                                    grossincome(m);
```

This makes the ECA-rules more convenient to write since we do not have to know what stored functions affect a derived function. ECA-rules in AMOS can also specify conjunctive events such as in the rule:

```
create rule check_new(department d) as
    for each employee e, manager m
    on updated(dept(e)) and
       updated(employee.netincome->number(e))
    when dept(e) = d and
         m = mgr(e) and
         employee.netincome->number(e) > netincome(m)
    do rollback;
```

This rule specifies that new employees at a certain department are not allowed to be given an immediate income that is higher than the manager; if this is the case, then the transaction is considered faulty and is rolled back.

It is possible to specify the event of creating an object as well:

```
create rule check_new(department d) as
    for each employee e, manager m
    on created(e) and
       updated(employee.netincome->number(e))
    when dept(e) = d and
         m = mgr(e) and
         employee.netincome->number(e) > netincome(m)
    do rollback;
```

This rule specifies that new employees at the company, i.e. employees who did not previously exist in the company database, are not allowed to be given a salary higher than their manager.

Examples of AMOSQL rules for the applications studied in chapter 2 can be found in chapter 5. Efficiency issues of active rules are discussed in chapter 6. Implementation details on the active rules are discussed in chapter 7.

4 Heterogeneous Data Management

4.1 DBMSs in Networks

In both CIM and telecommunication networks (as presented in chapter 2) the DBMSs will be connected in networks. The networks will most likely be *heterogeneous*, i.e. the nodes in the networks will perform different functions in the system and will probably contain different pieces of software and sometimes have different hardware. The databases in the network will reflect this heterogeneity by storing different data in different nodes, e.g. data that is needed locally by the functions provided by a particular node. Sometimes the nodes will contain different DBMS products with different storage structures. To support access of several nodes in the same queries there is a need for a heterogeneous database layer that can interact with the different heterogeneous nodes.

4.2 Distributed v.s. Multidatabases Database Systems

In *distributed database systems* [94] the goal is to show a global schema to the user to give the illusion of a single database system. This causes much overhead during database operations to keep all the databases in consistent states. Queries that reference distributed data need to be optimized to minimize a total cost which includes accessing remote data over a network.

In *multidatabase systems* [19][69] the goal of providing a global distributed view has been relaxed and the user is allowed to reference the individual nodes in the system directly. This simplifies the implementation and provides more autonomy to each node while at the same time forcing the user to make sure that the nodes are kept consistent. Multidatabases are often used as platforms for connecting heterogeneous databases (which could be either single node or distributed databases). In some cases a global schema can be split into several *sub-schemas* that each represents data that is stored on a subset of the nodes. Then the multidatabase can support maintenance of each sub-schema. The sub-schemas can sometimes overlap, i.e. nodes can have different data in different sub-schemas. Multidatabases usually provide optimization of multidatabase queries which may involve translating data from different formats into some standard format provided by the multidatabase (as well as considering communication costs). Often heterogeneous networks are hierarchical (such as in tele-

communication networks). The sub-schemas can then reflect different levels in the network hierarchy.

In a *mediator* architecture [131] such as AMOS a multidatabase is used for integrating various data sources with non-conforming data formats which are integrated for access through the multidatabase.

4.3 About Paper III

Paper III presents an overview of the AMOS multidatabase architecture. This is an early paper that introduces the AMOS mediator architecture. More work done on the AMOS mediator and multidatabase functionality is described in [42][130].

4.4 Active Multidatabase Systems

Active DBMSs in networks is nothing new, but most such systems (such as AMOS) do not support active rules that span over several databases. One major problem that has to be resolved is how to generate and monitor events between several databases. A major problem is how to compare events originating in different nodes (such as which event occurred before another) and how to combine these events into complex events. Some research on this topic is presented in [106].

Another problem in active multidatabase (or distributed) systems is that it can be expensive to ship events over the network when the rules that monitor the events are not executed by the same DBMS as where the events originate. One technique that can be used is to have the rule compiler split multidatabase rules (i.e. rules that have event expressions or conditions that reference changes to data stored in other databases) into several rules that are shipped and inserted into the databases where the different events originate. The original rule must then be defined to collect the results from all the distributed rules to finally determine if the multidatabase-rule has been triggered and if the condition is true.

Active multidatabases can be used for adding constraints to data stored in several heterogeneous database systems. In [27] a toolkit for constraint management in a heterogeneous information system is presented. In [119] a description of some work can be found on the specification of a language for achieving rule-based interoperabillity among heterogeneous systems. In [137] work can be found on maintaining consistency of an integrated view of information from various distributed data sources.

4.5 Heterogeneous Databases in CIM

In CIM applications there are many sources of information that are not directly involved in the manufacturing process, but which are still desirable to be able

to access from the CIM system. This can be information such as product data (e.g. what sub-parts a manufactured item consists of), inventory data (e.g. how many sub-parts are available in stock), economic data (e.g. profit margins in terms of how much each produced item should cost based on the sum of the costs of the sub-parts and the cost of manufacturing), and sales data (e.g. how many items should be produced).

For a CIM system to be able to access heterogeneous data a mediating DBMS such as AMOS can provide uniform access to all data (fig. 4.1). A mediating ADBMS can also support monitoring changes to heterogeneous data such as monitoring the number of parts in store and to order more when the stock runs low.

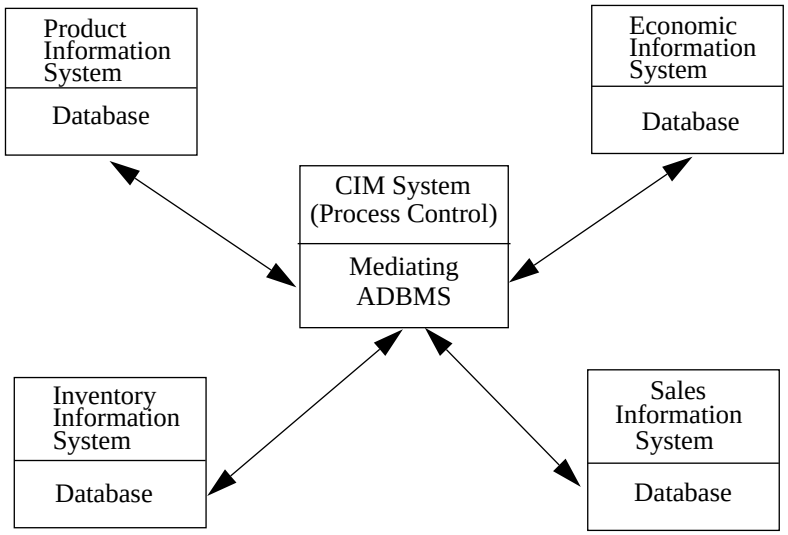


Figure 4.1: A mediating ADBMS in CIM for accessing heterogeneous data

Monitoring of heterogeneous databases usually requires some monitoring support from each involved DBMS. If ECA-rules or triggers are supported, these can be generated by the mediating ADBMS and be compiled into the involved databases (see section 9.5.8).

4.6 Heterogeneous Databases in Telecommunication Networks

In telecommunication networks there will be a need to integrate different heterogeneous databases such as DBMSs belonging to several parts of the network,

e.g. DBMSs in local networks and in public networks operated by different network providers.

Different kinds of heterogeneity can be defined on the basis of location, i.e. where the information is stored geographically, and on the basis of functionality, i.e. what functions in the network hierarchy are using the data (fig. 4.2). The higher up in the hierarchical distribution, the more the access sideways in the geographic distribution is utilized. In network traffic control there is little exchange of data between databases, usually only the data needed for setting up connections. In sectional and regional network management there can be a considerable exchange of data, e.g. passing of billing data between different parts of the network or network providers.

The DBMSs used in network control might be distributed to achieve high-performance transaction processing and to provide reliability at hardware failures [122]. The network management will collect data from the network control, such as basic billing data based on used resources. The data will be passed upwards in the hierarchy to off-line DBMSs for processing and, eventually, sending bills to subscribers. On-line billing might also be performed where

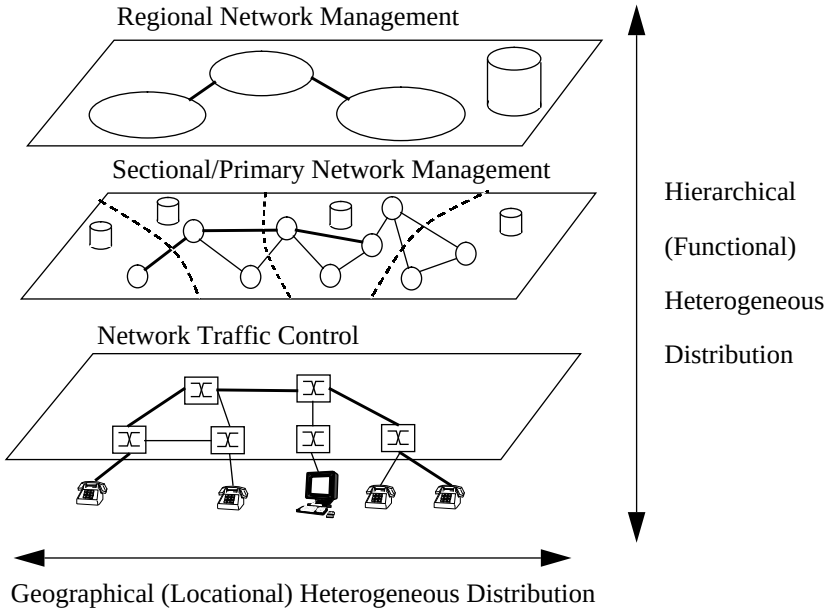


Figure 4.2: Heterogeneous dimensions in a network hierarchy

subscribers are notified of the cost while they are utilizing specific services.

Note that this is only a discussion of data passed between DBMSs in the network. Traffic data (e.g. speech data) in the physical network can be considerable, but this data usually originates from one user and is directly passed to another user. In cases where the databases contribute directly to the traffic data,

such as databases for network applications that send multi-media data to the network users, the heterogeneous distribution model has to be extended to include network applications.

By considering databases for network applications as well, we can define the generic term *network DBMS* as a DBMS that can manage the databases for network traffic control, network management, and network applications (fig. 4.3). Different network DBMSs can be used separately for each kind of database or sometimes for all kinds of databases at once. The most likely model is that several network DBMSs makes up a heterogeneous (multidatabase) DBMS architecture. The individual network DBMSs are most likely (homogeneous) distributed cluster DBMSs to provide the performance and high reliability required by the different database applications.

The configuration of the heterogeneous network DBMS depends on the geographical and hierarchical distribution of the network. The configuration of each network DBMS server (such as distribution topology, fragmentation and duplication of data) depends on the requirements of the specific database applications using a specific server.

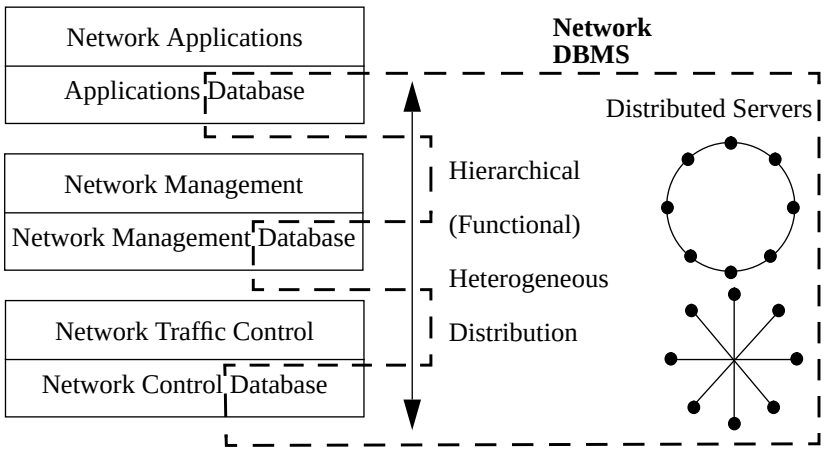


Figure 4.3: A network DBMS for managing databases of network control, network management, and network applications

Communication between heterogeneous DBMSs will be based on network specific protocols while communication between nodes in individual cluster nodes in each DBMS server can be any high-speed protocol (see section 9.5.3 for more discussion about protocols). The heterogeneous network DBMS will most likely have to mediate between various data sources, e.g. when mapping data from the physical level to a logical level in the network hierarchy, integrating databases managed by different network providers, or accessing databases belonging to specific network applications.

5 Applying Active Database Systems

5.1 Applications and Active Database Systems

In the development of new features in an active database system it is important to have some potential applications that require these features. In chapter 2 application studies of Computer Integrated Manufacturing (CIM) and telecommunications networks were presented. These studies were made to motivate some of the active functionality in AMOS. This chapter presents some possible scenarios of integration between the applications and ADBMS technology.

5.2 Scenarios for an ADBMSs in CIM Systems

Consider a CIM system for process control in a manufacturing plant. The actual control of the plant is carried out by a real-time process control system (fig. 5.1).

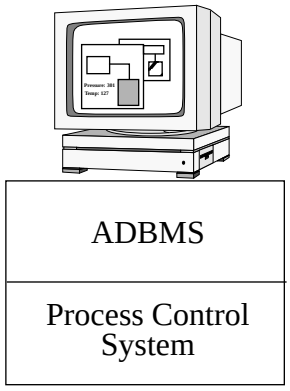


Figure 5.1: The Active DBMS for integration of a Process Control System

An ADBMS is used for storing data about plant layout (equipment such as manufacturing machines with actuators and sensors and their configuration),

equipment data (machine status, sensor readings), parts data (data about the parts being produced such as size, weight, colour, position, sub-parts, completeness status), and system configuration (configuration of the whole CIM system). The data from sensors is made available from the real-time process control system by either storing the data directly in the ADBMS or by allowing the ADBMS access the data as foreign data sources.

The ADBMS can monitor changes to sensor data through active rules by monitoring changes to sensor data stored in the database, or by letting the process control system send foreign events when a sensor has changed (every time or when there is a significant change). The active rules can, for example, be used for managing automatic redisplay functions in user interface tools [97] that display the status of the controlled plant or for detecting abnormal situations that the process control system cannot detect (such as situations involving several local control loops).

A Scenario for Automatic Redisplay of User Interface T ools

Let us take a scenario where an ADBMS should monitor the change of the state of a control process to automatically refresh interface tools that monitor the process. In the schema below the process state is defined as a foreign function that is exported by the process control system. The process control system also signals an update event when the state of the process changes. Note that it is up to the process control system to determine how often it should inform the ADBMS. It is very likely that the process state seen in the database might have a coarser granularity than the state used in controlling the process.

Interface tools are defined to have several interfaces that can be associated with a certain process that is being displayed. The interface management system provides the ADBMS with a function that can directly refresh a certain interface.

One active rule is defined that monitors updates to the state of a certain process and refreshes the corresponding interfaces of a given tool if the change exceeds some given threshold. If the interfaces are refreshed, then the rule caches the process state so that it can determine the change the next time the process state changes. The rule also automatically refreshes new interfaces associated with the monitored process and the given tool.

```
create type process;
create function process_state(process) -> real
as foreign;
create function cached_state(process) -> real;
create type tool;
create type interface;
create function tool_interface(tool, process) ->
bag of interface;
create function refresh(interface, real) -> boolean
as foreign;
```

```
create rule monitor_process(process p, real threshold,
                            tool t) as
for each real s
on update(process_state(p)) or added(tool_interface(t,p))
when abs(cached_state(p) - s) > threshold
     and s = process_state(p)
do  begin
      set cached_state(p) = s;
      for each interface i where i = tool_interface(t, p)
         refresh(i, s);
    end;
```

A Scenario for Monitoring Interaction Between Controlled Processes

Another use of an ADBMS in process control applications is for monitoring several processes at once. The control process system usually only monitors and controls each defined process separately. It usually has no way of determining interactions between the different processes. All such interactions were defined when the different processes were defined. An ADBMS can support adding new monitoring functionality that detects interaction between processes in the control system. Take an example of two robots that can in rare circumstances interact by entering each other's working areas. This would usually be avoided by the control system, but in case of software errors it can still occur. Two functions are used, one for detecting what is an illegal interaction and a procedure that calls the control system to resolve the conflict. The resolve function could decide to move one of the robots out of the way or to emergency stop both robots.

```
create type robot;
create function position(robot)
     -> <real x, real y, real z>;
/* The current 3D-position of the robot */
create function working_area(robot)
     -> <real origin, real radius>;
/* The working area of a robot defined as a sphere */
create function within(real x, real y, real z,
                       real origin, real radius)
     -> boolean as foreign;
/* Foreign function that checks if a point is inside
   a sphere */
create function illegal_interaction(robot r1,
                                    robot r2)
     -> boolean as
     select within(x1, y1, z1, or2, ra2) or
            within(x2, y2, z2, or1, ra1)
     for each real x1,real y1,real z1,real or1,real ra1,
```

```
              real x2,real y2,real z2,real or2,real ra2
     where <x1,y1,z1> = postion(r1) and
           <or1, ra1> = working_area(r1) and
           <x2,y2,z2> = postion(r2) and
           <or2, ra2> = working_area(r2);
create function resolve_interaction(robot r1,
                                    robot 2)
-> boolean as ...; /* Application dependent code */

create rule monitor_interaction(robot r1, robot r2) as
   on update(position(r1)) or
      update(position(r2))
   when illegal_interaction(r1, r2)
   do resolve_interaction(r1, r2);
```

This rule could also be written more declaratively as a CA-rule where the rule compiler will deduce what events to monitor from the condition. The following rule would monitor changes to robot positions as well as changes to the working areas.

```
create rule monitor_interaction(robot r1, robot r2) as
   when illegal_interaction(r1, r2)
   do resolve_interaction(r1, r2);
```

Automatic Generation of Active Rules

In a real CIM scenario it is not likely that the operators or engineers that set-up the applications are familiar with DBMSs or have the knowledge to use a query language or even less likely, active rules in an ADBMS. It is more likely that the CIM system will have a dedicated application task language for specifying the tasks of the control applications. The system can then compile the task programs into the schema definitions, queries, and active rules that are needed by the ADBMS.

An extension of the above scenarios above could be an architecture similar to ARAMIS (see Paper I) where a high-level task language is used for defining the manufacturing tasks which are then executed on top of the ADBMS with support of automatically generated active rules that interact with the real-time process control system (fig. 5.2). In this architecture the environment with the controlled equipment is directly modelled in a World Model (WM) stored in the database. The WM (as defined in Paper I) would consist of two parts, a high-level control part managed by the ADBMS and a low-level control part which managed by the real-time process control system.

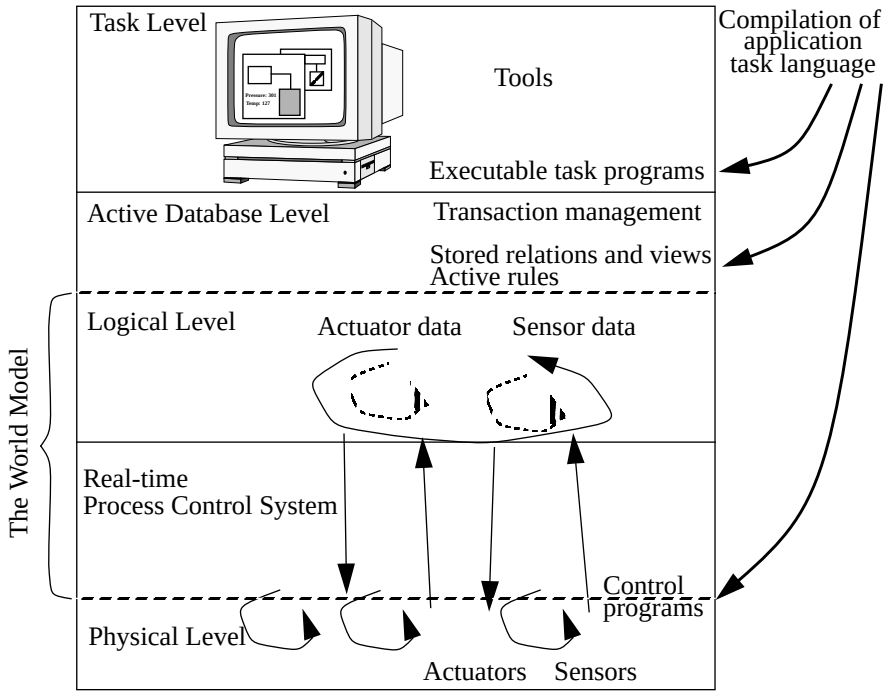


Figure 5.2: Compiling a task language into active rules

Here the physical control is still being performed by the real-time process control system, but the effects on the data in the WM from control loops are known by the ADBMS. If an actuator is ordered to change its physical state, then the ADBMS will know what the outcome in terms of sensor values will be (if the control was successful). This allows for defining constraint rules over allowable actuator settings in terms allowed sensor values.

The active rules can be involved in more coarse-grained control loops that monitor and affect several fine-grained control loops in the process control system. These may be rules that monitor the progress of the whole manufacturing process not just one operation.

5.3 About Paper IV

The ARAMIS system was taken as an application to study the use of the active rules in AMOS. The initial ideas of how this could be achieved are presented in Paper IV. An implementation was made that joined together the ideas in the ARAMIS architecture with AMOS. The results from this work are presented in [43][44].

The rules presented in this paper seem to be involved in fairly low level control, but the real-time control loops are in practice just initiated by the actions of these rules. The procedures called in the actions will generate calls to the underlying process control system which will schedule the activities to meet any real-time requirements.

5.4 About Paper V

In a manufacturing application there are usually different phases in the manufacturing process. Different operations are usually applicable in different phases and thus different active rules are also applicable. One result from the AMOS-ARAMIS study was the need for grouping rules. To support this the concept of *rule contexts* was developed and is presented in Paper V.

Rule contexts support grouping of rules to enable efficient activation and deactivation of several rules simultaneously. The rule contexts have been implemented in AMOS and further implementation details can be found in chapter 7.

5.5 Monitoring Long-running Transactions

Active rules and triggers have been used for organizing long-running transactions [32]. In *workflow management systems* [57] business or control processes are modelled using workflow languages that specify sequences and interactions of operations in the processes. Workflow management systems can use transactions and active rules in ADBMSs for organizing and synchronizing processes as long-running transactions. In AMOS support for long-running transactions have been implemented as *sagas* [51]. Sagas specify sequences of committed transactions that are chained together with compensating transactions that are executed when a saga is aborted. The rule contexts presented in Paper V can be attached to a saga to be automatically activated for transactions in that saga. When the saga is exited, i.e. when a transaction in that saga is committed, the attached rule contexts can be automatically checked and then be deactivated. Alternatively, the attached contexts can be checked when the complete saga is committed. When the saga is entered again, the attached rule contexts are automatically reactivated. When (or if) a saga is rolled back, the same or different rule contexts can be defined to be automatically activated and checked. See section 14.1 for syntax descriptions of sagas in AMOS.

5.6 Scenarios for ADBMSs in Telecommunication Networks

Future telecommunication networks will have very complicated monitoring tasks that need to be supported. Integration of DBMSs in the networks provides a dynamic property to data management that is needed for long-term network management and will support future growth and network reconfigurations. In

the development of these networks there are many challenges such as meeting performance requirements and supporting new functionality. Integrating ADBMSs with these networks is perhaps a solution that meets some of the challenges. Using an ADBMS to store network data makes it possible to monitor changes to the network data through the database.

The integration of ADBMSs must, however, be planned early in the development phases of the networks since it can radically change how the different functionality is implemented and where different data is generated, stored, and can be accessed. The use of ADBMSs can be divided into different scenarios for network traffic control, network management, and network applications.

5.6.1 ADBMSs in Telecommunications Network Traffic Control

Today's large telecommunication exchanges usually consist of a large number of software functions integrated with dedicated DBMSs for passive data management. The argument for using active DBMS technology in network traffic control is perhaps not as strong as for network management, but there are some possible uses here as well.

These systems have some functions for monitoring the state of the switch hardware and software. By storing state information in an ADBMS active rules can be used for monitoring the status of the system and to inform operators of possible problems and failures.

Some subscriber services in local exchanges could also be supported by the use of active rules, such as:

- wake-up call, where a temporal event triggers a wake-up call to a sleeping subscriber
- call diversion, where active rules trigger on an attempted call-setup to a busy subscriber and re-routes the call somewhere else
- call waiting, where active rules trigger on an attempted call-setup to a busy subscriber and notifies the called subscriber with an intrusion signal
- malicious call tracing, where a called subscriber triggers an active rule by pressing a button and that traces where the call comes from
- call cost information, where the termination of a call triggers a rule that causes call cost information to be sent to the calling (paying) subscriber

Subscriber services are quite intricate and will require many different triggering points, i.e. different events, to be defined during call-setup, during a call, and at call termination.

One main problem in these systems is that of handling *feature interactions* [60], i.e. how to handle the activation of several interacting subscriber services. Currently each manufacturer handles these in different ways. In heterogeneous telecommunication network products from many manufacturers this can be difficult without some cooperation based on standards.

One (among many) solutions of how to specify and perhaps resolve feature interaction is to specify logical rules that determine what should happen in specific interaction scenarios. These rules are then used during implementation of the features. Currently the features are often hard-coded into the systems, which makes it difficult if the interaction between the features is changed or if new features are added. A more flexible solution is to support direct execution of the rules in a rule-based system that is integrated into the systems. Rules in an ADBMS are possible candidates for achieving this. Below follows a simplified scenario to give the reader some idea of how some functionality for telecommunication network traffic control can be provided by an ADBMS.

A Scenario for using Active Rules in Subscriber Services

In this scenario an ADBMS is directly involved in the call set-up phase and can monitor the actions of the subscribers which are defined as instances of the type 'subscriber '. A call is, somewhat simplified, defined as an instance of the 'call' type and can be in the states: ringing, busy , connecting, connected, or disconnecting. A call has one subscriber who controls the call and one or several participants (fig. 5.3). The call controller is usually the one who initiated the call and is usually the one who is billed. In conference calls a participant can sometimes take over the role of controller , e.g. if the original controller hangs up. If a call participant has some special service, e.g. call diversion, then he is usually billed for the transferred call, not the caller .

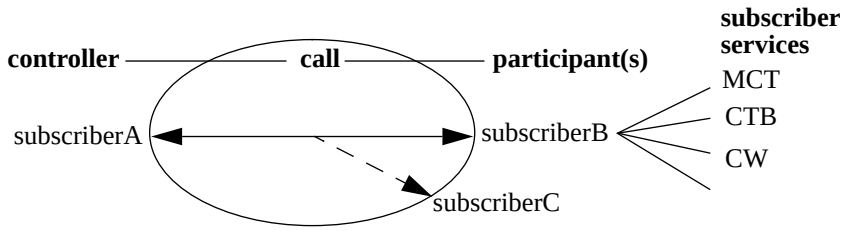


Figure 5.3: A simplified call model

Three rules are defined in this scenario:

- One rule that supports the malicious call tracing (MCT) where a subscriber can press the R(flash)-button when he/she receives an unwanted call. The rule will find who made the call and inform the operator of the malicious call.
- One rule that supports the call transfer (call diversion) on busy (CTB). The rule automatically transfers the caller to another number if the called subscriber is busy .
- One rule that supports the call waiting (CW) service where a subscriber who is engaged in a call will receive notification about incoming calls. The subscriber can choose to talk to the new calling subscriber (switching

back and forth) or engage in a three-party call. The CW service has precedence over CTB so the CW rule is activated with a higher priority than the CTB rule.

```
create type subscriber;
create function key_press(subscriber) -> integer;
/* 0 - 9, 10(*), 11(#) 12(R) */
create type subscriber_service;
create function provided_service(subscriber)
       -> bag of subcriber_service;

create type call;
create function call_controller(call) -> subscriber;
create function call_participant(call)
       -> bag of subscriber;
create function call_state(call) -> charstring;
/* ringing, busy, connecting, connected, disconnecting */

create type charging_record;
create function tariff(call) -> charging_record as ...
/* Procedure for calculating cost of a call */
create function tariff(subscriber_service)
       -> charging_record as ...
/* Procedure for calculating flat rate cost of a service */
create function tariff(subscriber_service, call)
       -> charging_record as ...
/* Procedure for calculating usage cost of a service in a
   call */
create function bill(subscriber s, charging_record cr)
       -> boolean as ...
/* Procedure that bills or prepares billing of subscriber
   depending on the state of the call in the charging
   record */

/* ***************************** */
/* Malicious Call Tracing Service */
/* ***************************** */

create subcriber_service instances :MCT;
create function malicious_call(subscriber sa,
                                  subscriber sb)
       -> boolean as ...
/* Procedure that informs operator about the MCT */

create rule MCT(subscriber sb) as
       for each call c, subscriber sa
```

```
       on added(key_press(sb))
       when key_press(sb) = 12 and /* Flash (R) */
            sb = call_participant(c)
            state(c) = “connected” and
            sa = call_controller(c) and
       do malicious_call(sa, sb); /* Inform operator */

create function setup_MCT(subscriber s) -> boolean as
       begin
            add provided_service(s) = :MCT;
            activate rule MCT(s);
            bill(s, tariff(:MCT)); /* Bill subscriber */
       end;

/* ******************** */
/* Call Transfer on Busy */
/* ******************** */

create subscriber_service instances :CTB;
create function ctb_redirect(subscriber sb)
       -> subscriber sc;
create function reconnect(call c, subscriber s)
       -> boolean as ...
/* Procedure redirects a call attempt, will set
   call_state(c) = “connected” if successful */

create rule CTB(subscriber sb) as
       for each call c
       on updated(call_state(c))
       when call_state(c) = “busy” and
            call_participant(c) = sb
       do begin
            reconnect(c, ctb_redirect(sb));
            bill(sb, tariff(:CTB, c));
          end;

create function setup_CTB(subscriber sb,
                          subscriber sc) -> boolean
       begin
            add provided_service(sb) = :CTB;
            set ctb_redirect(sb) = sc;
            activate rule CTB(sb) priority 3;
            bill(sb, tariff(:CTB)); /* Flat rate */
       end;
```

```
/* ************ */
/* Call Waiting */
/* ************ */

create subscriber_service instances :CW;
create function cw_inform(subscriber) -> boolean as ...
/* Procedure that sends intrusion signal to busy
   subscriber */

create rule CW(subscriber sb) as
       for each call c
       on updated(call_state(c))
       when call_state(c) = “busy” and
            call_participant(c) = sb
       do begin
            cw_inform(sb);
            bill(sb, tariff(:CW, c));
          end;

create function setup_CW(subscriber s) -> boolean
       begin
            add provided_service(s) = :CW;
            activate rule CW(s) priority 4;
            bill(s, tariff(:SW)); /* Flat rate */
       end;
```

Note that in this scenario the problem of feature interaction is handled through different rule priorities on the rule activations where the call waiting rule is given a higher priority than the call transfer on busy rule. This is a major simplification of the feature interaction problem in general. In a real scenario the interaction can be more complex where combinations of features can provide new functionality (e.g. such as initiating a three-party call or conference call when accepting new participants in call waiting) that has to be defined procedurally or with other rules.

5.6.2 Telecommunication Network Management

Management of future broad-band telecommunication networks such as ATM-networks is a more likely application for the use ADBMSs than in telecommunication network control. The network can be modelled in the ADBMS using international standards for network specification. An ADBMS with OO capability will directly be able to store OO specifications based on standards such as GDMO [74]. The ADBMSs will have to support the different interfaces for network management as specified in section 2.6.2 and support the mappings between the logical view and the physical views of the network. The ADBMS must also have a foreign data source interface supporting protocols such as

SNMP MIB (see section 9.5.3) for directly accessing data in different network elements. The monitoring functionality in SNMP MIB will have to be adapted to present changes to network elements as foreign events that the active rules can monitor.

A Scenario for Monitoring Failures in an A TM-network

In an ATM-network a connection is considered full-duplex (two data streams in two directions). If a failure occurs we have to define where it is detected and what has been af fected by the failure. In the *ATM-network failur e model* we distinguish between network elements upstream or downstream from the point of failure (fig. 5.4). A detected failure is not always detected at the point of failure, i.e. faulty equipment might be detected through lost packages downstream from the equipment. If failures are detected in both streams in a connection, but at dif ferent points, then the point of failure can sometimes be deduced by meeting half-way and upstream from the detected failures.

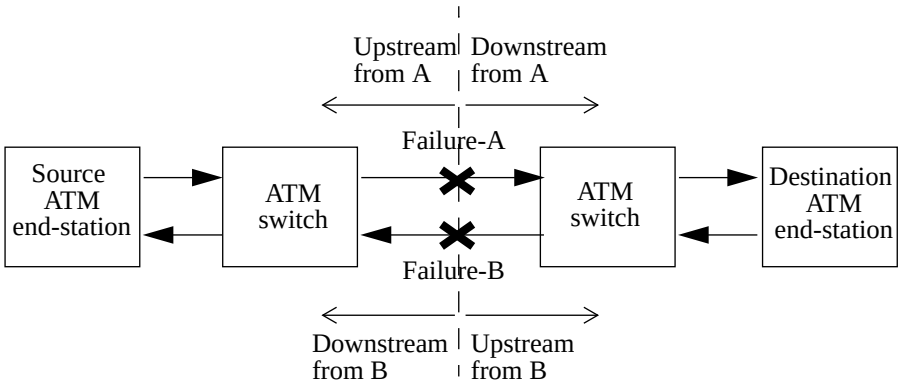


Figure 5.4: The ATM-network failure model

To explain how connections in an A TM-network can be monitored we also have to define an *ATM-network connection model* . A network connection is defined by a *trail* connection through the dif ferent subnetworks that make up the A TM-network (fig. 5.5). Connections between subnetworks are grouped into *links* with *connection termination points* (mapped to VCIs) in each end of the subnetwork. The trail is an allocated sequence of connections with *trail termination points* in both ends of the network.

The following database schema is an example of how the model above can be implemented in an ADBMS. It should be regarded as a somewhat simplified scenario [1] to show the complexity of the application and not as a full implemen-

1. The modelling is based on definitions from the ATM-Forum [3][81] and in GDMO [74].

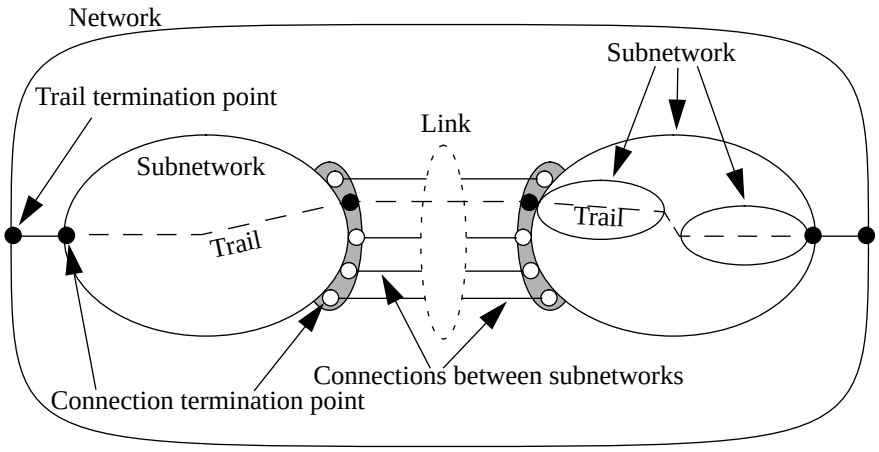


Figure 5.5: An ATM-network connection model

tation. Two rules are defined: one rule that monitors the *quality of service* (qos) in a specific trail connection and one that monitors alarms that affect a specific link. Note that most functions in this scenario would be foreign functions that access and monitor the network status externally from the database. Monitoring foreign functions is discussed further in chapter 9.

```
create type network;
create type subnetwork subtype of network;
create type link;
create function bandwidthUpstream(link) -> real;
create function bandwidthDownstream(link) -> real;
/* Quality of service */
create function qosUpstream(link)
                        -> <real error_ratio,
                            real loss_ratio,
                            real average_delay,
                            real variance_delay,
                            real misinsertion_rate>;
create function qosDownstream(link)
                        -> <real error_ratio,
                            real loss_ratio,
                            real average_delay,
                            real variance_delay,
                            real misinsertion_rate>;
create function unacceptableQos(
                        real error_ratio,
                        real loss_ratio,
                        real average_delay,
```

```
                        real variance_delay,
                        real misinsertion_rate)
       -> boolean as ...
/* Function that checks if qos is too low */
create type connection;
/* Traffic data */
create function receiveData(connection)
              -> <real bandwidth,
                  real average_information_rate,
                  real peak_information_rate,
                  real burstiness>;
create function transmitData(connection)
              -> <real bandwidth,
                  real average_information_rate,
                  real peak_information_rate,
                  real burstiness>;

create type TP; /* Termination Point */
create type linkTP subtype of TP;
create type connectionTP subtype of TP;
create type trailTP subtype of TP;
create type managedElement;
create type equipment subtype of managedElement;
create type software subtype of managedElement;

create function networkLink(subnetwork, subnetwork)
                        -> link;
create function linkConnections(link)
                        -> bag of connection;
create function connectionEnds(connection)
                        -> <connectionTP, connectionTP>;
create function direction(TP)
       -> chartstring; /* Uni- or bi-directional */
create function trailConnections(trail)
                        -> bag of connection;
create function networkElement(network)
                        -> bag of managedElement;
create function alarm(TP) -> bag of charstring;
create function alarm(managedElement)
                        -> bag of charstring;
create function raiseTrailAlarm(trail, charstring)
       -> boolean as ...
/* Procedure that signals trail alarm */
create function raiseLinkAlarm(link, charstring)
       -> boolean as ...
/* Procedure that signals link alarm */
```

```
create rule monitorQos(trail t) as
       for each connection c, link l,
                real er, /* error_ratio */
                real lr, /* loss_ratio */
                real ad, /* average_delay */
                real vd, /* variance_delay */
                real mr /*misinsertion_rate */
       on update(qosUpstream(l)) or
          update(qosDownstream(l))
       when c = trailConnections(t) and
            c = linkConnections(l) and
            (<er, lr, ad, vd, mr> = qosUpstream(l) or
             <er, lr, ad, vd, mr> = qosDownstream(l)) and
            unacceptableThroughput(er, lr, ad, vd, mr)
       do begin
            raiseLinkAlarm(l, “Unacceptable QOS”);
            raiseTrailAlarm(t, “Unacceptable QOS”);
          end;

create rule monitorLink(link l) as
       for each connectionTP ctp, connectionTP ctp1,
                managedElement me, connection c,
                subnetwork sn, subnetwork sn1
       on added(alarm(ctp)) or added(alarm(me))
       when c = linkConnections(l) and
            (<ctp, ctp1> = connectionEnds(c) or
             <ctp1, ctp> = connectionEnds(c)) and
            (networkLink(sn, sn1) = l or
             networkLink(sn1, sn) = l) and
            me = networkElement(sn)
       do for each charstring ad
          where (ad = alarm(ctp) or ad = alarm(me))
          raiseLinkAlarm(l, ad);
```

5.6.3 Telecommunication Network Applications

In network applications the use of a DBMS in general is fairly obvious (see section 2.6.3), while the use of an ADBMS, however, is perhaps not as obvious. Some uses of active database functionality in telecommunication network applications could be services such as:

- subscribing to newsgroups that are of particular interest to the user, e.g. subscribing to news relating to the user’s professional interests or hobbies
- integration of vehicle navigation systems and mobile telecommunication networks

where an ADBMS monitors the position of vehicles (e.g. using GPS) and informs drivers of the routes to destinations or changes to the traffic situation (e.g. by messages to a mobile terminal)

- on-line monitoring of access to certain services, e.g. can be used by the users to keep track on misuse of their account and for the service providers to monitor user access profiles (of different users, at different hours)
- on-line billing by monitoring the use and total cost of the services used so far, e.g. to help the user monitor how much his or her family has spent so far and perhaps to lock certain services in order to avoid receiving unexpectedly high bills

A Scenario for On-line Billing of Network Applications

This scenario relates to the network control scenario where charging records are produced from the use of various services. Here charging records are being received (monitored) that are the result of the use of some network application. Here an ADBMS is intended to perform on-line billing by incrementally calculating the bill so far and sending it to the user (perhaps showing up in a counter or as a meter on the display on his cellular phone) while he is using the service. When the user finishes the call or the application session, the bill is finalized and sent to the user as an invoice. Alternatively, the incremental bill can directly be used to decrease some virtual resources (e.g. NetCash [89]) of the user while he is using the application.

```
create function charging_records(subscriber)
       -> bag of charging_record;
create type bill;
create function subscriber_bill(subscriber) -> bill;
create function calculate_bill
       (subscriber, bill, charging_record) -> bill;
/* Procedure that incrementally calculates the
   subscriber's bill */
create function notify_subscriber(subscriber, bill)
       -> boolean as ...
/* Procedure that informs subscriber of current bill
   amount */

create rule online_billing(subscriber s) as
    on added(charging_records(s))
    do /* EA-rule */
    begin
      set subscriber_bill(s) =
          calculate_bill(s, added(charging_records(s)),
                                 subscriber_bill(s));
      notify_subscriber(s, subscriber_bill(s));
    end;
```

Note that in this rule the added charging record data is accessed in the action of the rule to incrementally calculate the new bill. This is an access of an event function outside the event part of a rule (in the condition or the action) which is discussed in section 7.4.

6 Efficient Rule Execution Using Partial Differencing

6.1 Efficiency Problems in ADBMSs

One major requirement that was concluded from the case studies in chapter 2 was the need for efficient execution of rules with complex conditions. When introducing rules into a database it is crucial that the overall performance of the DBMS is not impaired significantly. *Rule monitoring* is the activity of monitoring changes to the state of rule conditions. A *naive* method of detecting changes is to execute the complete condition when an event that triggers the rule has occurred. This, however, can be very costly, since a rule condition can span over large portions of the database. One major reason for introducing rules into databases is that it is more efficient to detect changes to the data inside the database than to have applications pose queries that detect the changes. When rules are introduced, they will impose some overhead on transactions that are performing updates to data referenced in event specifications or conditions of active rules. Since these transactions might belong to an application unaware of the rules it affects, it is important that the rules are processed efficiently.

6.2 Partial Differencing of Rule Conditions

This chapter presents a technique for efficient evaluation of rule conditions. The technique is based on incremental evaluation techniques and is named *partial differencing*. It is used for efficient monitoring of active rules in AMOS. The technique is especially designed for *deferred* rules, i.e. rules where the rule execution is deferred until a check phase that usually occurs when transactions are committed. The technique can also be used for immediate rule processing [31].

Partial differencing is presented here based on a conference paper [110] (Paper VI) along with some continued work. A *difference calculus* is defined for computations of the changes to the result of database queries and views. Queries and relational views are regarded as functions over sets of tuples and the calculus for monitoring changes is regarded as an extension of set algebra. Let P be a function dependent on the functions Q and R, denoted the *influents* of the *affected* function P. The problem of *finite differencing* [80][96] is how to calculate the changes to P, ΔP, in terms of the changes to its influents. With *par-*

tial differencing, changes to P are defined as the combination of the changes to P originating in the changes to each of its influents. Thus, ΔP is defined in terms of the *partial differential* functions ΔP/ΔQ and ΔP/ΔR. We will define how to automatically derive the partial differentials ΔP/ΔQ and ΔP/ΔR, and how to calculate ΔP from them. The calculus is mapped to relational algebra by defining partial differentials for the basic relational operators. Partial differencing has the following properties compared to other approaches:

- We assume that the number of updates in a transaction is usually small and often very few (or only one) tables are updated. Therefore, very few partial differentials are affected in each transaction. Each partial differential generated by the rule compiler is a relatively simple database query which is optimized using traditional query optimization techniques [107]. The optimizer assumes few changes to a single influent.

- We separately define *positive* and *negative* partial differentials, denoted $\Delta P/\Delta_+ Q$ and $\Delta P/\Delta_- Q$, respectively, since monitored conditions are often only dependent on insertions in influents (not on deletions), as will be shown. Furthermore, the partial differentials for handling insertions and deletions do not have the same structure. Conditions that depend on deletions are actually historical queries that must be executed in the database state when the deleted data were present. This makes negative differentials different and not easily mixable with positive ones.

- The calculus allows us to optimize both space and time. Space optimization is achieved since the calculus and the algorithm does not presuppose materialization of monitored conditions to find their previous state. Instead it gives a choice between materialization and computation of the old state from the new one, given all the state changes. Time optimization is achieved through incremental evaluation techniques.

- Based on the calculus, an algorithm has been developed for efficient rule condition monitoring by propagation of incremental changes through a *propagation network*. For correct handling of deletions in the absence of materializations and for efficient execution, a *breadth-first, bottom-up propagation* is made through the network of both insertions and (only when applicable) deletions. The algorithm reduces memory utilization by only temporarily saving the intermediate changes occurring during the propagation.

- For explainability, one can easily determine which influents actually caused a rule to trigger and whether it was triggered by an insertion or a deletion. It is straightforward to determine this by remembering which partial differentials were actually executed in the triggering.

Partial differencing has been implemented for CA-rules in AMOS and performance measurements have been made. We have implemented both our incremental algorithm and a ‘naive’ condition monitoring algorithm that recomputes the whole rule condition every time an update has been made to an influent affect-

ing a condition. The performance evaluation shows that for transactions with few updates our incremental algorithm scales better over the database size than the naive method. For transactions with many updates to several influents the method is not as efficient as the naive evaluation, but only by a factor that is constant over the size of the database.

The method supports ECA-rules as well; the event part just further restricts when the condition is tested. Partial differencing also allows for specifying events such as added, removed, and updated over views by propagating physical changes to the event parts of ECA and EA-rules (while propagating logical changes to conditions of CA and ECA-rules). Partial differencing for ECA-rules is discussed in section 6.15, section 6.16, and section 6.17.

6.3 Related Work

In [96] *finite differencing* was presented as a technique for improving the efficiency of the set-oriented programming language SETL. It was based on program transformations using differentiation operators defined for the basic set-functions in SETL. Finite differencing for maintaining derived data in materialized views in a functional data model was defined in [80]. In [13] finite differencing was used for maintaining materialized views in the relational data model defined in terms of Select-Project-Join (SPJ) views.

The technique was adopted for rule condition monitoring in HiPAC [31][103], Ariel [63], PARADISER [33], and in A-RDL [40][133]. Recent work on incremental maintenance of materialized views can be found in [61][62][75][77][100]. Related work on incremental evaluation of Datalog programs can be found in [36] and on change computation in deductive databases in [126].

In [103] incremental evaluation of SPJ-views was presented for efficient evaluation of ECA-rules with complex rule conditions. The work was based on defining an algebra for computations over database changes, Δ-relations. Each relation had an associated Δ-relation where the tuples that got added and deleted during database updates were stored. Each SPJ-view also had Δ-relations which were computed through a *chain-rule* for SPJ queries.

Our work differs from the above in that we deal with the problem of *partial differencing* of database queries, i.e. automatic generation of several separate partial differentials from a given rule condition rather than one large incremental expression. Furthermore, we also deal with deletions and incremental evaluation of deferred rule conditions.

In [99] the relational algebra is extended with incremental expressions. In [9] a method is presented that derives two optimized conditions, *Previously True* and *Previously False*, based on a materialization of a simple truth value of a condition. Since our rules are set-oriented, we need to consider sets of truth values.

A classical algorithm for incremental evaluation of rule conditions in AI is the Rete algorithm [49]. It is used to incrementally evaluate rule conditions (called patterns) in the OPS5 [17] expert system shell. OPS5 is a forward-chaining production rule system where all patterns are checked using Rete. Thus, in difference to active

database systems, all the instantiations of all patterns in an OPS5 program are incrementally maintained. Regular demand-driven database queries are not supported. In Rete the system records each incremental change (insertions or deletions, called tokens) to the stored data. For patterns that reference other patterns (i.e derived patterns) a *propagation network* is built that incrementally maintains the instances of the derived patterns. The propagation network may contain both selections (represented as *alpha nodes*) and joins (represented as *beta nodes*). The alpha nodes (selections) are always propagated before the beta nodes (joins).

The main problem with using Rete for rule matching in active databases is that Rete is very space inefficient for large databases since Rete saves all intermediate results for all rule conditions. Rete furthermore does not do join optimizations which may result in a combinatorical explosion of the size of the working memory [91]. To improve the performance of Rete the TREAT [91] algorithm was developed. TREAT avoids the combinatorical explosion by using relational database optimization techniques and has been shown to be more efficient for large databases than Rete [129].

Ariel [63] uses an extension of TREAT, A-TREAT, that further reduces the memory usage by avoiding to materialize some intermediate results by defining some selection nodes in the propagation network as simple relational expressions (named *virtual alpha nodes*). A related approach is proposed in [40] where an algorithm is presented that can take a set of rules and return a set of relational expressions that are the most profitable to materialize to support efficient execution of the rules. These are examples of how to trade query execution time for space in rule condition checking.

In contrast to the work above we use a propagation algorithm based on our calculus for partial differencing. The nodes in the propagation network do not reflect hard-coded primitive operations such as alpha or beta nodes, but represent temporary storage of data propagated from the nodes below. The arcs represent differential relational expressions that calculate the changes from an input node below that should be propagated to the output node above. By using *breadth-first, bottom-up* propagation to correctly and efficiently propagate both positive and negative changes without retaining space consuming materializations of intermediate views our algorithm differs from the PF-algorithm [65]. The materialized views can be very large and can even be considerably larger than the original database, e.g. where cartesian products or unions are used. This may exhaust memory or buffers when many conditions are monitored and the database is large.

In A-RDL [40][133] incremental evaluation and a *fixpoint* technique are used for determining whether a rule has been triggered and whether a set of rules will terminate, i.e. if they have some fixpoint. This technique is used in a rule system outside a DBMS where the client-server communication is intercepted to detect events. This is different from the approach in this thesis in which the rule condition monitoring is integrated into an ADBMS and where incremental expressions are generated that are executed by the query processor (i.e. by the ObjectLog interpreter). The fixpoint technique could be used as an extension to partial differencing, but this would require an analysis of the rule actions to determine what events will be generated, which is not discussed in this thesis.

In [75] a differential technique is presented for supporting efficient execution of

historical queries based on transaction time. The technique is based on efficient calculation of cached queries by using *differential files* that contain transaction time-ordered insert, delete, and update data. The differential files can be used for both *incremental calculation* (moving to a future state in the database) and *decremental calculation* (moving to a past state in the database). The technique is supported by differential versions of the relational operators (select, project, and join) which are used in full differential expressions similarly as in finite differencing. The differential expressions are optimized using a state transition network, dynamic programming techniques, and a number of optimization rules. The optimization technique is related to that in AMOS which is discussed further in section 6.10. The technique for supporting transaction time is related to an extension of partial differencing to support propagation of temporal information such as the time of updates. It is based on event histories ordered by transaction time for an integration of partial differencing and event propagation for ECA-rules. This makes it possible to use the same propagation network to calculate complex event specifications and to incrementally calculate complex rule conditions. This is more discussed further in sections 6.16 and 6.17.

In [22][117] ECA-rules are used to incrementally maintain materialized views. In [22] a technique is presented how the rules can be semi-automatically derived given the views to be materialized. The generated ECA-rules are parameterized to allow for a simple form of incremental evaluation. In [117] ECA-rules are used to manually maintain materialized views.

Heraclitus [50] is a dedicated database programming language which directly supports incremental evaluation by supporting *deltas*, i.e. objects containing update information. The deltas can be explicitly constructed, combined, and accessed through the programming language. The Heraclitus paradigm can be used to implement different execution models for active rules in an ADBMS.

In [93] a performance test is presented for incremental updates in two different rule-based programs. The first is the game of LIFE where incremental updates of a matrix of varying size are monitored. The second is a combinatorial optimization problem for allocating mortgage-backed securities. The results favour incremental update for the second program, but not for the first one. No real in-depth analysis is provided as to why this is so, only that updates in the first program produce major changes in the chain of inference which is unsuitable for incremental evaluation. In section 6.9 partial differencing is analyzed through a performance measurement consisting of seven different benchmarks.

In section 6.16 the propagation technique used for partial differencing is extended for propagating events of ECA-rules. ECA-rules in AMOS allow specifying the events added, removed, and updated on both stored functions (tables) and derived functions (views). This is related to work on specifying *composite events* in Sentinel [25], SAMOS [54], and Ode [56] which are defined in terms of primitive events. The propagation techniques for calculating the occurrence of composite events from primitive events are related to the propagation technique presented in this thesis. The propagation techniques for composite event detection is usually based on techniques and data structures specialized for event detection and not for change propagation. In [25] an *event*

tree is used for propagating events, [54] uses a *modified colored Petri Net*, and [56] uses a *state automata*. The work in this thesis focuses on detecting changes to relational views, i.e. implicit composite event specifications of the events added, removed, and updated. The technique could be extended to allow explicit composite event specifications and of other events as well, but this is outside the scope of this thesis.

6.4 An Example Rule with Efficiency Problems

Let us look at the inventory rule example from chapter 3 again. When the quantity of an item drops below a certain threshold, new items are to be automatically ordered. Here is the `monitor_items` rule again:

```
create type item;
create type supplier;
create function quantity(item) -> integer;
create function max_stock(item) -> integer;
create function min_stock(item) -> integer;
create function consume_frequency(item)
            -> integer;
create function supplies(supplier) -> item;
create function delivery_time(item,supplier)
            -> integer;
create function threshold(item i) -> integer
    as
    select consume_frequency(i) *
       delivery_time(i, s) + min_stock(i)
    for each supplier s where supplies(s) = i;

create rule monitor_items() as
     when for each item i
     where quantity(i) < threshold(i)
     do order(i,max_stock(i) - quantity(i));
```

Executing this rule can be very inefficient if the database contains thousands of items. If we evaluate the condition as it stands we will scan the quantity and calculate the threshold for all items every time there is a change to some item. The user could define a CA-rule that is parameterized with specific items (`monitor_item`), but if we want to monitor all items we would have to activate this rule for every item. Alternatively, an ECA-rule could be defined that captures the relevant events for a specific item and passes the item to the condition through a shared variable. This requires that the user knows what events can affect the condition. The rule above is more elegant since the user does not need to know what events the rule should monitor, thus we want the ADBMS to efficiently monitor these kind of CA-rules as well.

6.5 CA-rule Semantics and Function Monitoring

Another requirement that was concluded from the case studies presented in chapter 2 was the need for transparent access and monitoring of external data. The AMOS data model uses functions that return data of some specific type when accessing all data. When the AMOS data model was extended with rules it was natural to use functions when monitoring changes to data as well. The semantics of the rules in AMOS is thus based on function monitoring [102]. To be more specific, rules are based on the *when-function-changes-do-procedure* semantics (fig. 6.1).

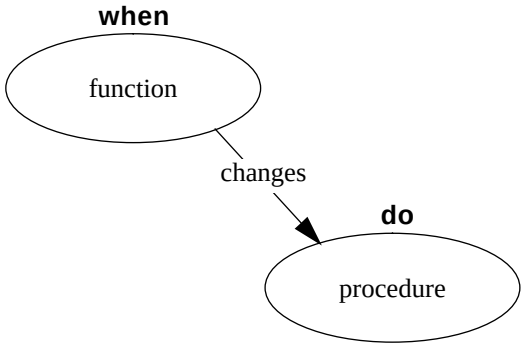


Figure 6.1: AMOSQL rule semantics

Take a CA-rule r(x) defined as **when** c(x) **do** a(x).

This is a *forward chaining* rule that means: when there is a change in the database that might change the value c(x), then 'execute a(x) when c(x) is evaluated to be true'. This is an imprecise definition of rule semantics, one really has to separate between *strict* and *nervous* rule semantics. Strict rule semantics for r would really be 'execute a(x) when c(x) is evaluated to be true after previously being false' and nervous rule semantics would be 'execute a(x) whenever c(x) is evaluated to be true regardless of whether it was true before'.

In order to explain how a rule is transformed into a function and a procedure, a new notation is introduced.

The rules are written as:

<name>(<parameter-specification>) =
[<variable-quantification>] (<condition> ⇒ <action>)

where the condition is a functional expression and the action a procedural expression.

Functions are written as:

<name>(<parameter-specification>) = [<variable-quantification>]
select <return-specification>
where <logical-expression>

and procedures as:

<name>(<parameter-specification>) = <procedure statements>

All parameters and heads of functions are subscripted with type information that specifies the types of the incoming parameters and the types of the returned values of functions, respectively.

We can now write the rule r as:

$$r(x_{\text{type of } x}) = (c(x) \Rightarrow a(x)),$$

where $c(x)$ is a function call that returns a boolean value, i.e. $c(x_{\text{type of } x})_{\text{boolean}}$, and where $a(x)$ is a procedure call. Note that x will be bound when the rule is activated.

Next we define a *condition function* f_c that returns the data shared by the condition and action (here data of the type of x):

$$f_c(x_{\text{type of } x})_{\text{type of } x} = \text{select } x \text{ where } c(x)$$

This function returns a set of values of type x for all $c(x)$ that return true. We also define an *action procedure* f_a that takes the output of the condition function (here the type of x) as argument and executes the action statements:

$$f_a(x_{\text{type of } x}) = a(x),$$

Rule condition monitoring is now defined as function monitoring of f_c, i.e. monitoring of changes to the set of values that f_c returns. Executing CA-rules can now be seen as the function application $f_a(f_c(x))$. To be more precise, rule execution is defined for *nervous* rule behaviour as executing f_a on all the changes to f_c, Δf_c:[1]

$$\forall\, x_{\text{type of } x} \text{ where } x \in \Delta f_c(x)$$
$$\text{do } f_a(x)$$

We also define *strict* rule behaviour as executing f_a on all the changes of f_c, Δf_c, only if they are not present in the old state of f_c, $(f_c)_{old}$, which is the value of f_c the last time this particular rule activation was checked. Strict rule monitoring is defined as:

$$\forall\, x_{\text{type of } x} \text{ where } x \in \Delta f_c(x) \wedge x \notin (f_c)_{old}(x)$$
$$\text{do } f_a(x)$$

Note that before the action procedure of a triggered rule is executed, a conflict resolution method is applied. The condition of a rule can contain any logical expression and the action any logical expressions as well as side effects. For a rule

1. Actually we execute the action procedure f_a on the positive changes representing additions ($\Delta_+ f_c$) to f_c,

$r(x_{\text{type of x}}) = \forall\, y_{\text{type of y}}\ (c1(x) \wedge c2(y) \Rightarrow a1(x) \wedge a2(y))$,

and where $c1(x)$ and $c2(y)$ are boolean functions, i.e. $c1(x_{\text{type of x}})_{\text{boolean}}$ and $c2(y_{\text{type of y}})_{\text{boolean}}$. The condition function to monitor is defined as:

$$f_c(x_{\text{type of x}})_{<\text{type of x, type of y}>} = \forall\, y_{\text{type of y}}\ \text{select x, y where } c1(x) \wedge c2(y),$$

and the action procedure to execute is defined as:

$$f_a(x_{\text{type of x}}, y_{\text{type of y}}) = a1(x) \wedge a2(y).$$

The semantics of nervous and strict rule execution, respectively, are defined as:

$$\forall\, x_{\text{type of x}}, y_{\text{type of y}} \text{ where } <x, y> \in \Delta f_c(x)$$
$$\text{do } f_a(x, y)$$

$$\forall\, x_{\text{type of x}}, y_{\text{type of y}} \text{ where } <x, y> \in \Delta f_c(x) \wedge <x, y> \notin (f_c)_{old}(x)$$
$$\text{do } f_a(x, y)$$

Note that here x is bound when the rule is activated, but y is free and is fetched from the database.

Since functions are defined semantically as representing sets of values the rules are said to have set-oriented semantics[1], i.e. the rules monitor changes of a set that represents the condition and executes the action on each element of the set that represents the changes to the condition set.

Some rules do not use the set-oriented semantics, as is the case with constraint rules that have actions that do transaction roll-backs. Such rules do not use any explicit values that have been produced in the condition when executing the action. Constraint rules are defined as:

$$r(x_{\text{type of x}}) = (c(x) \Rightarrow \text{rollback}),$$
$$f_c(x_{\text{type of x}})_{\text{boolean}} = \text{select true where } c(x),$$
$$f_a() = \text{rollback},$$

The condition function f returns true if $c(x)$ returns a non-empty answer and false otherwise. The semantics is defined for nervous and strict rule execution, respectively, by:

$$\forall\, x_{\text{type of x}} \text{ where } \Delta f_c(x)$$
$$\text{do } f_a()$$

$$\forall\, x_{\text{type of x}} \text{ where } \Delta f_c(x) \wedge \neg(f_c)_{old}(x)$$
$$\text{do } f_a()$$

1. To support the same set-oriented action execution as in Starburst [132] the action function f_a should take a set of all the changes from Δf_c instead of applying $f_a(x)$ for each possible x.

Actually, strict rule semantics does not make sense for a rule that does a rollback since the rollback will probably undo the changes and make the condition false. However, if the action of a constraint rule just signals the user or application that a constraint was violated then we need strict rule semantics since we usually do not want to signal more than once.

Since rules are objects of the type ‘rule’, the rule activation can be defined as a procedure:

$$\text{activate}(r_{\text{rule}}, l_{\text{list of object}})$$

where r is a rule object and l is a list of objects that r is parameterized by. For rule activations with explicit priorities[1] the activate procedure is overloaded as

$$\text{activate}(r_{\text{rule}}, l_{\text{list of object}}, p_{\text{integer}})$$

where p is the priority of the rule activation. Rule deactivation is defined likewise.

In the inventory rule example the rule compiler generates the condition function `cnd_monitor_items` from the condition of the rule `monitor_items`. This function returns all the items with quantities below the threshold. Condition monitoring is done by monitoring changes to the condition function `cnd_monitor_items`.

```
create function cnd_monitor_items() -> item
    as
    select i for each item i
    where quantity(i) < threshold(i);
```

The action part of the rule generates a procedure that takes an item as argument and orders new items to fill the inventory.

```
create function act_monitor_items(item i)
    -> boolean[2] as
    order(i, max_stock(i) - quantity(i));
```

At run-time the `act_monitor_items` procedure will be applied to the set of *changes* calculated from the differential denoted Δ`cnd_monitor_items`.

AMOSQL is a stream-oriented language so the action procedure is executed for every changed value of the condition. With strict semantics the action procedure is executed *only* when the truth value of the monitored condition changes from false to true in some transaction. With nervous semantics the rule sometimes triggers when there has been an update that causes the rule condition to become true without having been false previously. Nervous semantics is often sufficient; however, in our example strict semantics is preferable since we only want to order an item once when it

1. Rule activations without explicit priorities are given the lowest priority.
2. A procedure that does not explicitly return anything implicitly returns a boolean.

becomes low in stock. Note that before the action part of a triggered rule is executed a conflict resolution method based on priorities is applied.

6.6 ObjectLog

AMOSQL functions are compiled into an intermediate language called ObjectLog [82]. ObjectLog is inspired by Datalog [21][125] and $\mathcal{LDL}$ [28], but provides new facilities for effective processing of OO queries. ObjectLog supports a type hierarchy, late binding, update semantics, and foreign predicates.

- Predicate arguments are *objects*, where each object belongs to one or more *types* organized in a type hierarchy that corresponds to the type hierarchy of AMOS.
- Object creation and deletion semantics maintain the referential integrity of the type hierarchy.
- Update semantics of predicates preserve the type integrity of arguments. The optimizer relies on this to avoid dynamic type checking in queries.
- Predicates can be overloaded on the types of their arguments and results.
- Predicates can be further overloaded on the binding patterns of their arguments, i.e. on which arguments are bound or free when the predicate is evaluated.
- Predicates can be not only facts and Horn clause rules, but also optimized calls to invertible *foreign predicates* implemented in a procedural language. In the current system foreign predicates can be written in C and Lisp.
- Predicates themselves as well as types are objects, and there are second order predicates that produce or can be applied to other predicates. Second order predicates are crucial for late binding and recursion.

The translation from AMOSQL to ObjectLog consists of several steps (fig. 6.2). The *flattener* transforms AMOSQL `select` statements into a flattened `select` statement where nested functional calls have been removed by introducing intermediate variables. The *type checker* annotates functions with their type signatures in the *type adornment phase*, and finds the actual functions for overloaded functions (in case of early binding) or adds dynamic type checks (in case of late binding) in the *overload resolution phase*. The ObjectLog generator transforms stored functions into facts and derived functions are transformed into Horn clause rules. The *ObjectLog generator* also translates foreign functions into foreign predicates. The *ObjectLog optimizer* finally optimizes ObjectLog programs using cost-based optimization techniques. More about the translation steps and the optimization techniques can be found in [48][82]. The optimization step is discussed further in section 6.10.

The ObjectLog facts represent base relations, i.e. tables, and the Horn Clauses represent derived relations, i.e. views. In section 14.2 in the appendix the relationship between Datalog (which can be seen as a relational subset of ObjectLog) and the relational operators is presented. The AMOSQL query compiler will

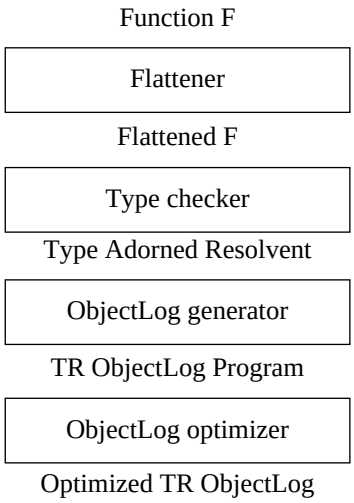


Figure 6.2: The translation of AMOSQL to ObjectLog

translate the stored functions defined in the example into facts and the condition function and the threshold function will be compiled into the following ObjectLog Horn Clauses:

```
cnd_monitor_items_item(I) ←
  quantity_item,integer(I,_G1) ∧
  threshold_item,integer(I,_G2) ∧
  _G1 < _G2

threshold_item,integer(I,T) ←
  consume_frequency_item,integer(I,_G1) ∧
  delivery_time_item,supplier,integer(I,_G2,_G3) ∧
  supplies_supplier, item(_G2, I) ∧
  _G4 = _G1 * _G3 ∧
  min_stock_item,integer(I,_G5) ∧
  T = _G4 + _G5
```

By looking at the generated ObjectLog for `cnd_monitor_items` and `threshold` we can define a dependency network (fig. 6.3) that specifies what changes can affect the differential Δ`cnd_monitor_items`. Each edge in the dependency network defines the influence from one function to another. With each edge we also associate the partial differentials that calculate the actual influence from a particular node. For instance, Δ`quantity` is an influent of Δ`cnd_monitor_items` with a partial differential Δ`cnd_monitor_items`/Δ`quantity` (the edge marked * in fig. 6.3). The dependency network is con-

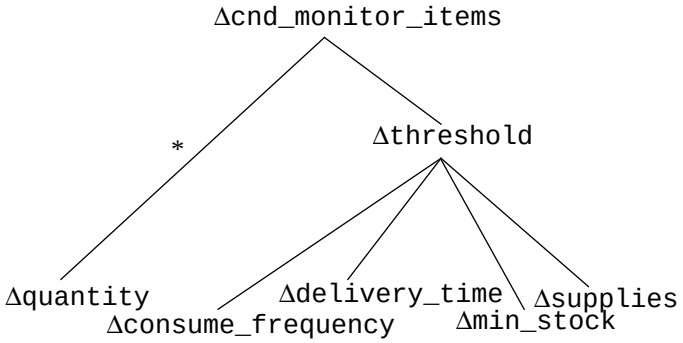


Figure 6.3: Dependency network of the rule condition

structed from the definition of the condition function and its sub-functions. By analyzing the ObjectLog code instead of the AMOSQL the rule compiler can find all dependencies, i.e. all relations that can affect a rule condition. The ObjectLog relations above are not optimized; this is a later stage which generates optimized and executable ObjectLog code. The ObjectLog code is used to generate partial differentials (which are optimized by the ObjectLog optimizer) and the propagation network (see section 6.7.3).

The transformations that are presented for partial differencing can be made on either un-optimized or optimized ObjectLog programs. The resulting ObjectLog programs will need to be re-optimized in any case.

6.7 The Calculus of Partial Differencing

The calculus of partial differencing is our theoretical basis for incremental evaluation of rule conditions. It formalizes update event detection and incremental change monitoring. The calculus is based on the usual set operators *union* ($\cup$), *intersection* ($\cap$), *difference* (-), and *complement* (~). The boolean algebra of ObjectLog can directly be translated[1] into set algebra which is easier to comprehend than relational algebra. Three new operators are introduced, *delta-plus* (Δ_+), *delta-minus* (Δ_-), and *delta-union* ($\cup_\Delta$). Δ_+ returns all tuples added to a set over a specified period of time, and Δ_- all tuples removed from the set. A *delta-set* (Δ-set) is defined as a disjoint pair $<\Delta_+S, \Delta_-S>$ for some set S and $\cup_\Delta$ is defined as the union of two Δ-sets. The calculus is general and in section 6.7.6 partial differencing of the relational algebra operators is shown.

Separate *partial differentials* are generated for monitoring insertions and dele-

1. There exists an isomorphism f: $<O, \neg, \wedge, \vee> \rightarrow <2^{At(O)}, \sim, \cap, \cup>$ between the boolean algebra and set algebra [1] where O is the domain of objects in the database, $\neg$ is negation based on the Closed World Assumption, $\wedge$ is logical conjunction, $\vee$ is logical disjunction, $2^{At(O)}$ is the power set of atoms in O, ~ is set complement, $\cap$ is set intersection, and $\cup$ is set union.

tions for each influent of a derived relation. The intuition is to calculate positive partial differentials (monitoring insertions) in the new state of the database. The negative partial differentials (monitoring deletions) are calculated in the old state since this was when the deleted tuples were present in the database.

The old state of a relation is calculated from the new state by performing a *logical rollback* (similar to decremental computation in [75]) that inverts the changes to the database. Given the value of S_{new} we can calculate S_{old} by inverting all operations done to S, i.e. by using $S_{old} = (S_{new} \cup \Delta\_S) - \Delta_+S$. The calculus is based on accumulating all the relevant updates to base relations during a transaction. These accumulated changes are then used to calculate the partial differentials of derived relations. Changes are propagated in a breadth-first, bottom-up manner through a propagation network where the Δ-sets can be seen as temporary 'wave-front' materializations. Calculating the old state, S_{old}, requires all the propagated changes that influence S, i.e. the complete Δ_+S and Δ_S, which in turn requires a breadth-first, bottom-up propagation algorithm.

The algorithm guarantees that all changes to influents of an affected relation are propagated before the changes to the affected relation are propagated further. Therefore, by propagating breadth-first, bottom-up we can calculate the old states (S_{old}) of relations by doing a logical rollback. Next we define how to accumulate these changes and how to generate partial differentials.

6.7.1 Differencing of Base Relations

All changes to base relations, i.e. stored functions, are logged in a logical undo/redo log. During database transactions and before the physical update events are written to the log, a check is made as to whether a stored base relation was updated that might change the truth value of some activated rule condition. If so, the *physical events* are accumulated in a Δ-*set* that reflects all *logical events* so far of the updated relation. Only those functions that are influents of some rule condition need Δ-sets. The Δ-sets can be discarded when the changes of the affected relations have been calculated, which saves space. Since CA-rules are only triggered by logical events the physical events have to be added with the *delta union* operator, $\cup_\Delta$, that cancels counteracting insertions and deletions in the Δ-set. The Δ-set for a base relation B is defined as:

$\Delta B = <\Delta_+B, \Delta\_B>$,

where Δ_+B is the set of added tuples to B and Δ_B is the set of removed tuples, they are defined as:

$\Delta_+B = B - B_{old}$ and[1]
$\Delta\_B = B_{old} - B$, and thus
$B_{old} = (B \cup \Delta\_B) - \Delta_+B$

We define $\cup_\Delta$ formally as:

$$\Delta B_1 \cup_\Delta \Delta B_2 = <(\Delta_+B_1 \cup \Delta_+B_2) - (\Delta\_B_1 \cup \Delta\_B_2), (\Delta\_B_1 \cup \Delta\_B_2) - (\Delta_+B_1 \cup \Delta_+B_2) >$$

1. The current database always reflects the new state.

The operator ensures that we only consider the net-effect of updates to a function. Updates to stored functions are made by first removing the old value tuples and then adding the new ones. For example, let us update the minimum stock of some item twice assuming that `min_stock` was originally 100:

```
set min_stock(:item1) = 150;
set min_stock(:item1) = 100;
```

This produces the physical update events:

```
-<min_stock,:item1,100>,
+<min_stock,:item1,150>,
-<min_stock,:item1,150>,
+<min_stock,:item1,100>.
```

The Δ-set for `min_stock` changes accordingly with:

```
Δmin_stock = <{},{<:item1,100>}>
Δmin_stock = <{<:item1,150>},{<:item1,100>}>
Δmin_stock = <{},{<:item1,100>}>
Δmin_stock = <{},{}>
```

i.e. there is no net effect of the updates.

6.7.2 Partial Differencing of Views

As for base relations, the Δ-set of a relational view, i.e. a derived function, is defined as a pair:

$\Delta P = <\Delta_+P, \Delta_-P>$

We need to define how to calculate the Δ-set of an affected view in terms of the Δ-sets of its influents. To motivate our calculus we next exemplify change monitoring of views for positive changes (adding) and negative changes (removing), respectively. We then show how to combine partial differentials into the final calculus.

6.7.3 Positive Partial Differentials

For a view P defined as a Horn Clause with a conjunctive body, let I_p be the set of all its influents. The positive partial differentials $\Delta P/\Delta_+X_i$, $X_i \in I_p$ (for insertions only) are constructed by substituting X_i in P with its positive differential Δ_+X_i. The full positive differential is found by performing a $\cup_\Delta$ of all the positive partial differentials.

For example, if we have

```
p(X, Z) ←
        q(X, Y) ∧
        r(Y, Z)
```

then

```
Δp(X, Z)/Δ+q ←
        Δ+q(X, Y) ∧
        r(Y, Z)
```

and

Δp(X, Z)/Δ_+r $\leftarrow$
q(X, Y) $\wedge$
Δ_+r(Y, Z)

If DB_{old} consists of the stored relations (facts) q(1, 1), r(1, 2), r(2, 3), then we can derive p(1, 2).
A transaction performs the updates
assert q(1, 2), assert r(1, 4)

DB_{new} now becomes q(1, 1), q(1, 2), r(1, 2), r(1, 4), r(2, 3), and we can derive p(1, 2), p(1, 3), p(1, 4)

The updates produce the Δ-sets,
Δq = <{<1,2>},{}>
Δr = <{<1,4>},{}>

Then Δp(X, Z)/Δ_+q = <{<1,3>},{}>
and Δp(X, Z)/Δ_+r = <{<1,4>},{}>
and joining with $\cup_\Delta$ finally gives
Δp = <{<1,3>,<1,4>},{}>

The AMOSQL compiler expands as many derived relations as possible to have more degrees of freedom for optimizations. The condition function of our running example will be expanded to:

```
cnd_monitor_items_item(I) ←
  quantity_item,integer(I,_G1) ∧
  consume_frequency_item,integer(I,_G2) ∧
  delivery_time_item,supplier,integer(I,_G3,_G4) ∧
  supplies_supplier, item(_G3, I) ∧
  _G5 = _G2 * _G4 ∧
  min_stock_item,integer(I,_G6) ∧
  _G7 = _G5 + _G6 ∧
  _G1 < _G7
```

The positive partial differentials based on the influents quantity and consume_frequency are defined as:

```
Δcnd_monitor_items_item(I)/Δ+quantity ←
  Δ+quantity_item,integer(I,_G1) ∧
  consume_frequency_item,integer(I,_G2) ∧
  delivery_time_item,supplier,integer(I,_G3,_G4) ∧
  supplies_supplier, item(_G3, I) ∧
  _G5 = _G2 * _G4 ∧
  min_stock_item,integer(I,_G6) ∧
```

```
  _G7 = _G5 + _G6 ∧
  _G1 < _G7

Δcnd_monitor_items_item(I)/Δ+consume_frequency ←
  quantity_item,integer(I,_G1) ∧
  Δ+consume_frequency_item,integer(I,_G2) ∧
  delivery_time_item,supplier,integer(I,_G3,_G4) ∧
  supplies_supplier, item(_G3, I) ∧
  _G5 = _G2 * _G4 ∧
  min_stock_item,integer(I,_G6) ∧
  _G7 = _G5 + _G6 ∧
  _G1 < _G7
```

The other differentials Δcnd_monitor_items/Δ+delivery_time, Δcnd_monitor_items/Δ+supplies, and Δcnd_monitor_items/Δ+min_stock are defined likewise. Using these partial differentials we can build a *propagation network* for cnd_monitor_items (fig. 6.4). This is

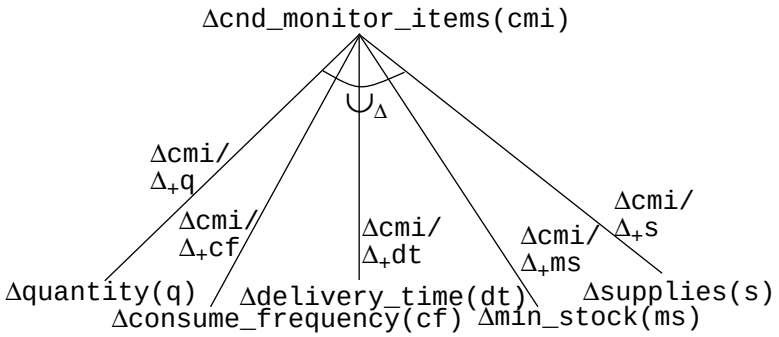


Figure 6.4: Propagation network of the rule condition

basically the dependency network (fig. 6.3) augmented with partial differentials. One difference to fig. 6.3 is that the propagation network for cnd_monitor_items is flat since the AMOS query compiler expands functions as far as possible. In the case of late binding this is not possible and the result is a more bushy network (see section 6.13). In section 6.10.3 we show how sub-expressions can be reused to produce a more bushy network that enables node sharing.

Note that the examples above only deal with conjunctions in the bodies of the Horn Clauses. In ObjectLog disjunctions are introduced in the body only and not as separate Horn Clauses as in traditional Datalog[1]. Disjunctions, i.e. unions, are treated

1. In ObjectLog separate Horn Clauses are generated for different AMOSQL functions that are overloaded on the type signatures of a single function name. Since only one of these functions is chosen at run-time, this is not a disjunction.

in section 6.7.5

6.7.4 Negative Partial Differentials

Often the rule condition depends only on positive changes, as for the `monitor_items` rule. However, for negation and aggregation operators, negative changes must be propagated as well. For strict rule semantics, propagation of negative changes is also necessary for rules whose conditions are negatively affected by other rules' actions, i.e. for a rule defined to be strict it is necessary to monitor the negative partial differentials as well.

In our example in section 6.7.3 the two partial differentials of the relation P with regard to the negative changes of Q and R are defined as:

$$\Delta p(X, Z)/\Delta\_q \leftarrow \Delta\_q(X, Y) \wedge r_{old}(Y, Z)$$

and

$$\Delta p(X, Z)/\Delta\_r \leftarrow q_{old}(X, Y) \wedge \Delta\_r(Y, Z)$$

where $R_{old} = (R \cup \Delta\_R) - \Delta_+R$ and where Q_{old} is defined likewise.

These can be calculated by a logical rollback (fig. 6.5) or be maintained by materialization.

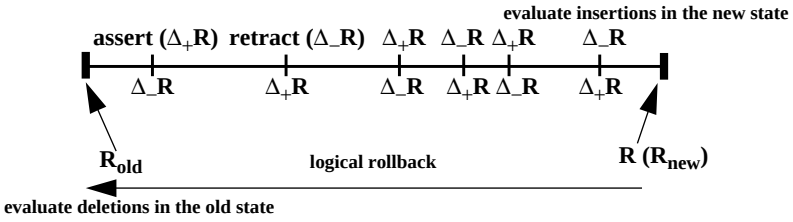


Figure 6.5: Calculating the old state by a logical rollback

The materializations can be space-consuming so doing a logical rollback will save space. The execution cost (in time or in number of operations) of performing one logical rollback of a relation is comparable to the execution cost of maintaining a materialization of that relation (i.e. the execution cost of updating the materialization). If a logical rollback of a relation is performed several times during the rule checking phase it can, however, cost more than maintaining the materialization.

The full negative differential is found by performing a $\cup_\Delta$ of all the negative partial differentials and the complete differential is found by a $\cup_\Delta$ of all

the positive and negative partial differentials.

Let DB_{old} consist of the stored relations (facts) `q(1, 1)`, `r(1, 2)`, `r(2, 3)`, from `p` defined above we can now derive `p(1, 2)`. A transaction performs the updates:

```
assert q(1, 2), assert r(1, 4),
retract r(1, 2), retract r(2, 3)
```

DB_{new} is now `q(1, 1)`, `q(1, 2)`, `r(1, 4)`, and we can derive `p(1, 4)`. The updates give the Δ-sets,

$\Delta q = \langle\{\langle 1,2\rangle\},\{\}\rangle$

$\Delta r = \langle\{\langle 1,4\rangle\},\{\langle 1,2\rangle,\langle 2,3\rangle\}\rangle$.

Then $\Delta p(X, Z)/\Delta_+ q = \langle\{\},\{\}\rangle$,

$\Delta p(X, Z)/\Delta_+ r = \langle\{\langle 1,4\rangle\},\{\}\rangle$,

$\Delta p(X, Z)/\Delta_- r = \langle\{\},\{\langle 1,2\rangle\}\rangle$,

and joining with $\cup_\Delta$ gives

$\Delta p = \langle\{\langle 1,4\rangle\},\{\langle 1,2\rangle\}\rangle$.

Note that if we did not use the old state of `q` (q_{old}) in $\Delta p(X,Z)/\Delta_- r$ we would get

$\Delta p = \langle\{\langle 1,4\rangle\},\{\langle 1,2\rangle,\langle 1,3\rangle\}\rangle$, which is clearly wrong.

6.7.5 The Calculus of Partial Differentials

Let $\Delta_+ P$ be the set of additions (positive changes) to a view P and $\Delta_- P$ the set of deletions (negative changes) from P. As before, the Δ-set of P, ΔP, is a pair of the positive and the negative changes of P:

$$\Delta P = \langle \Delta_+ P, \Delta_- P \rangle$$

As for base relations, we formally define the *delta-union*, $\cup_\Delta$, over differentials as:

$$\Delta P_1 \cup_\Delta \Delta P_2 = \langle (\Delta_+ P_1 \cup \Delta_+ P_2) - (\Delta_- P_1 \cup \Delta_- P_2), \\ (\Delta_- P_1 \cup \Delta_- P_2) - (\Delta_+ P_1 \cup \Delta_+ P_2) \rangle$$

Next we define the *partial differential*, ΔP/ΔX, that incrementally monitors changes to P from changes of each influent X. *Partial differencing* of a relation is defined as generating partial differentials for all the influents of the relation. The net changes of the partial differentials are accumulated (using $\cup_\Delta$) into ΔP.

Let I_p be the set of all relations that P depends on. The Δ-set of P, ΔP, is then defined by:

$$\Delta P = \cup_\Delta \frac{\Delta P}{\Delta X} \overset{1}{=} \cup_\Delta \left\langle \frac{\Delta P}{\Delta_+ X}, \frac{\Delta P}{\Delta_- X} \right\rangle, \forall X \in I_p$$

For example, if P depends on the relations Q and R then:

1. Equivalent to $\cup_\Delta \Delta P/\Delta X$.

$$\Delta P = \frac{\Delta P}{\Delta Q} \cup_\Delta \frac{\Delta P}{\Delta R} = <\frac{\Delta P}{\Delta_+ Q}, \frac{\Delta P}{\Delta_- Q}> \cup_\Delta <\frac{\Delta P}{\Delta_+ R}, \frac{\Delta P}{\Delta_- R}>$$

To detect changes of derived relations we define intersection (conjunction), union (disjunction), and complement (negation) in terms of their differentials as:

$$\Delta(Q \cap R) = <(\Delta_+ Q \cap R) \cup (Q \cap \Delta_+ R), \{\}> \cup_\Delta <\{\}, (\Delta_- Q \cap R_{old}) \cup (Q_{old} \cap \Delta_- R>$$

$$\Delta(Q \cup R) = <(\Delta_+ Q - R_{old}) \cup (\Delta_+ R - Q_{old}), \{\}> \cup_\Delta <\{\}, (\Delta_- Q - R) \cup (\Delta_- R - Q)>$$

$$\Delta(\sim Q) = <\Delta_- Q, \Delta_+ Q>$$

From the expressions above we can easily generate the simpler expressions in the case of, for instance insertions only. For example, when only considering insertions, changes to intersections are defined as:

$$\Delta(Q \cap R) = <(\Delta_+ Q \cap R) \cup (Q \cap \Delta_+ R), \{\}>$$

6.7.6 Partial Differencing of the Relational Operators

The calculus of partial differencing can easily be applied to the relational algebra to incrementally evaluate its operators. This is illustrated by table 6.1. This

P	$\frac{\Delta P}{\Delta_+ Q}$	$\frac{\Delta P}{\Delta_+ R}$	$\frac{\Delta P}{\Delta_- Q}$	$\frac{\Delta P}{\Delta_- R}$
$\sigma_{cond} Q$	$\sigma_{cond} \Delta_+ Q$		$\sigma_{cond} \Delta_- Q$	
$\pi_{attr} Q$	$\pi_{attr} \Delta_+ Q$		$\pi_{attr} \Delta_- Q$	
$Q \cup R$	$\Delta_+ Q - R_{old}$	$\Delta_+ R - Q_{old}$	$\Delta_- Q - R$	$\Delta_- R - Q$
$Q - R$*	$\Delta_+ Q - R$	$Q \cap \Delta_- R$	$\Delta_- Q - R_{old}$	$Q_{old} \cap \Delta_+ R$
$Q \times R$	$\Delta_+ Q \times R$	$Q \times \Delta_+ R$	$\Delta_- Q \times R_{old}$	$Q_{old} \times \Delta_- R$
$Q \bowtie R$	$\Delta_+ Q \bowtie R$	$Q \bowtie \Delta_+ R$	$\Delta_- Q \bowtie R_{old}$	$Q_{old} \bowtie \Delta_- R$
$Q \cap R$	$\Delta_+ Q \cap R$	$Q \cap \Delta_+ R$	$\Delta_- Q \cap R_{old}$	$Q_{old} \cap \Delta_- R$

Table 6.1: Partial differencing of the Relational Operators
*) $Q - R = Q \cap \sim R$

was generated by separating the expressions above for insertions and deletions and by using the definitions of the relational operators in terms of set operations. The table has been derived by applying the definitions in section 6.7.5 on the boolean expressions that represent the relational operators (see section 14.2

in the appendix). Note that the definitions in table 6.1 assume set-oriented semantics.

To make it easier to compare how partial differencing relates to other relational incremental evaluation techniques defined using relational algebra here is a relational version of the `cnd_monitor_items` condition function.

Let us assume we have the tables ITEMS and SUPPLIES:

ITEMS(INO, QUANTITY , MAX_ ST OCK, MIN_ST OCK,
CONSUME_FREQUENCY)
SUPPLIES(SNO, INO, DELIVER Y_TIME)

where INO and SNO are primary keys (in AMOS these are defined using the OIDs of the objects of the types item and supplier).

The CND_MONIT OR_ITEMS condition can now be defined by (in SQL syntax):

```
SELECT  INO
FROM ITEMS, SUPPLIES
WHERE ITEMS.INO = SUPPLIES.INO AND
        QUANTITY <
        CONSUME_FREQUENCY * DELIVER   Y_TIME + MIN_ST  OCK
```

This relational view can be defined using the standard relational operators in the query tree in fig. 6.6.

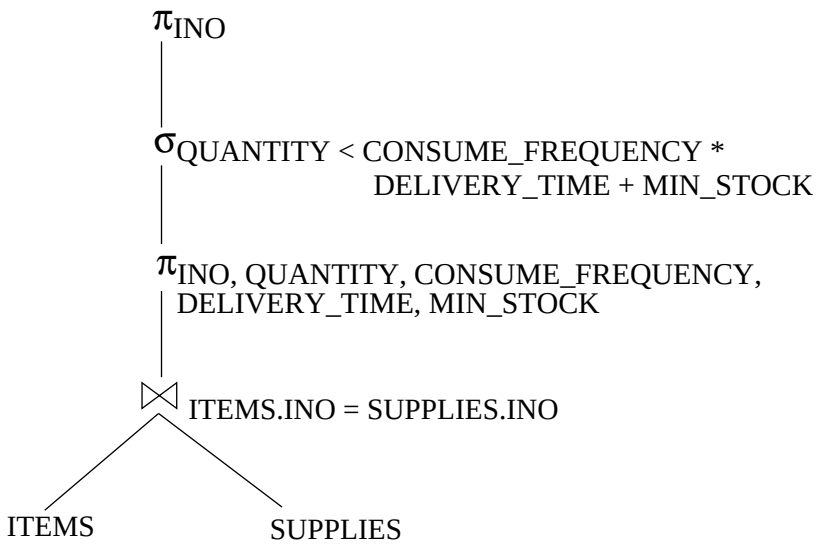


Figure 6.6: The relational operators for CND_MONITOR_ITEMS

This modelling dif fers slightly from the AMOSQL version since we only have

two tables compared to five stored functions. Here the functions are modelled as five attributes:

ITEMS.QUANTITY,
ITEMS.CONSUME_FREQUENCY,
ITEMS.MIN_STOCK,
SUPPLIES.INO, and
SUPPLIES.DELIVERY_TIME

Using table 6.1 we can see that the join gives two major partial differentials:

ΔCND_MONITOR_ITEMS/Δ_+ITEMS and
ΔCND_MONITOR_ITEMS/Δ_+SUPPLIES,
but since all non-key attributes can be directly updated these are really five partial differentials:

ΔCND_MONITOR_ITEMS/Δ_+ITEMS.QUANTITY,
ΔCND_MONITOR_ITEMS/Δ_+ITEMS.CONSUME_FREQUENCY,
ΔCND_MONITOR_ITEMS/Δ_+ITEMS.MIN_STOCK,
ΔCND_MONITOR_ITEMS/Δ_+SUPPLIES.INO, and
ΔCND_MONITOR_ITEMS/Δ_+SUPPLIES.DELIVERY_TIME

In these partial differentials the influent table and one of its attributes would be substituted by their differential versions where the differential of a table contains the changed rows and the differential of an attribute contains just the primary key and the changed value of the attribute. For example, for ΔCND_MONITOR_ITEMS/Δ_+ITEMS.QUANTITY we get the query tree in fig. 6.7 and for ΔCND_MONITOR_ITEMS/Δ_+SUPPLIES.INO we get the

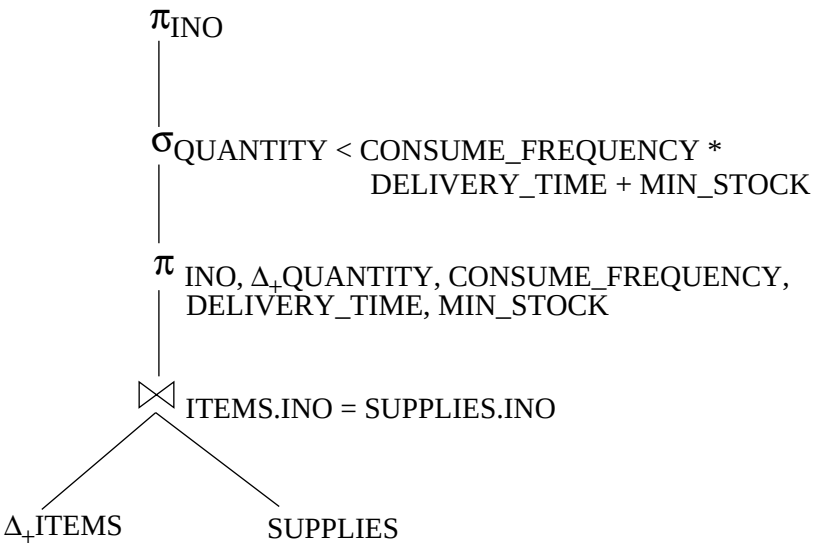


Figure 6.7: The relational operators for ΔCND_MONITOR_ITEMS/ Δ_+ITEMS.QUANTITY

query tree in fig. 6.8.

.

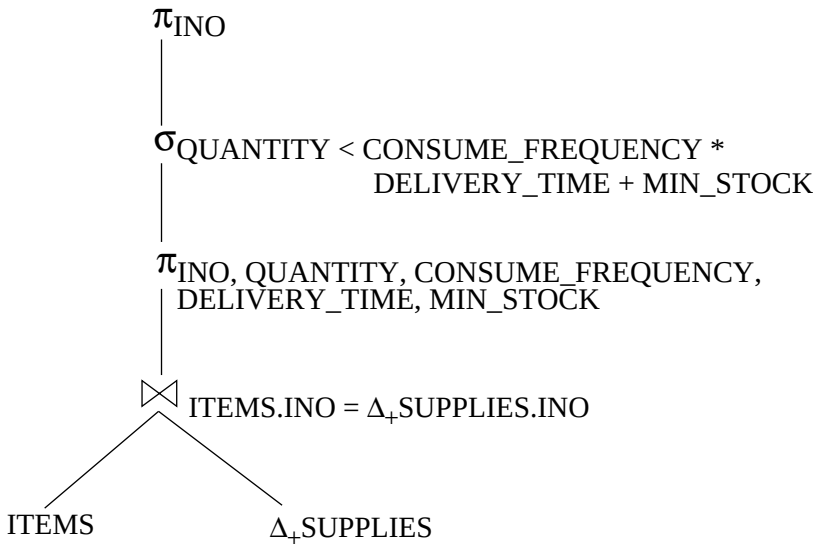


Figure 6.8: The relational operators for ΔCND_MONITOR_ITEMS/Δ_+SUPPLIES.INO

6.7.7 Using Partial Differencing for Maintaining Materialized Views

There exist many publications where incremental evaluation techniques are used for maintaining materialized views [80][13][62][75][77]. Partial differencing was developed for change monitoring of active rule conditions, but it can also be used for managing materialized views. The calculus of partial differencing is defined over Δ-sets which are defined as a pair consisting of a positive part (insertions) and a negative part (deletions). By viewing the materialized view as the positive[1] part of a Δ-set the calculus can be used to incrementally calculate the new state of a cached (materialized) view P, P', using P' = (P $\cup$ Δ_+P) - Δ_P. Likewise, we can also decrementally calculate the old value P of a view P', using P = (P' $\cup$ Δ_P) - Δ_+P. Using partial differencing we can calculate incremental expressions to calculate the new value of a materialized view. Here are the definitions of the new values of materialized views defined in terms of select (σ), project (π), and join ($\bowtie$) using the definitions in partial differencing and some simple transformations:

1. This is based on the assumption that the database only stores positive data, i.e. inserted positive facts. Databases allowing negated data are not considered in this thesis.

$(\sigma_{cond}P)' = \Delta_+(<\sigma_{cond}P, \{\}> \cup_\Delta \sigma_{cond}\Delta P) = \Delta_+(<\sigma_{cond}P, \{\}> \cup_\Delta (<\sigma_{cond}\Delta_+P, \{\}> \cup_\Delta <\{\}, \sigma_{cond}\Delta\_P>)) =$
/* Transform the second $\cup_\Delta$ using the definition of $\cup_\Delta$ */
$\Delta_+(<\sigma_{cond}P, \{\}> \cup_\Delta <\sigma_{cond}\Delta_+P, \sigma_{cond}\Delta\_P>) =$
/* Transform the last $\cup_\Delta$ */
$\Delta_+<(\sigma_{cond}P \cup \sigma_{cond}\Delta_+P) - \sigma_{cond}\Delta\_P,$
$\sigma_{cond}\Delta\_P - (\sigma_{cond}P \cup \sigma_{cond}\Delta_+P)> =$
/* Extract the Δ_+ part of the Δ-set */
$(\sigma_{cond}P \cup \sigma_{cond}\Delta_+P) - \sigma_{cond}\Delta\_P$

Note that $\sigma_{cond}P$ is here directly available since this is the view that is being maintained as a materialized view.

$(\pi_{attr}P)' = \Delta_+(<\pi_{attr}P, \{\}> \cup_\Delta \pi_{attr}\Delta P) = \Delta_+(<\pi_{attr}P, \{\}> \cup_\Delta (<\pi_{attr}\Delta_+P, \{\}> \cup_\Delta <\{\}, \pi_{attr}\Delta\_P>)) =$
/* Transform the second $\cup_\Delta$ using the definition of $\cup_\Delta$ */
$\Delta_+(<\pi_{attr}P, \{\}> \cup_\Delta <\pi_{attr}\Delta_+P, \pi_{attr}\Delta\_P>) =$
/* Transform the last $\cup_\Delta$ */
$\Delta_+<(\pi_{attr}P \cup \pi_{attr}\Delta_+P) - \pi_{attr}\Delta\_P,$
$\pi_{attr}\Delta\_P - (\pi_{attr}P \cup \pi_{attr}\Delta_+P)> =$
/* Extract the Δ_+ part of the Δ-set */
$(\pi_{attr}P \cup \pi_{attr}\Delta_+P) - \pi_{attr}\Delta\_P$

Note that $\pi_{attr}P$ is here directly available since this is the view that is being maintained as a materialized view.

$(P \bowtie Q)' = \Delta_+(<P \bowtie Q, \{\}> \cup_\Delta \Delta(P \bowtie Q)) =$
$\Delta_+(<P \bowtie Q, \{\}> \cup_\Delta (<\Delta_+(P \bowtie Q), \{\}> \cup_\Delta <\{\}, \Delta\_(P \bowtie Q)>)) =$
/* Expand $\Delta_+(P \bowtie Q)$ and $\Delta\_(P \bowtie Q)$ using table 6.1[1] */
$\Delta_+(<P \bowtie Q, \{\}> \cup_\Delta$
$(<(\Delta_+P \bowtie Q') \cup (P' \bowtie \Delta_+Q), \{\}> \cup_\Delta$
$<\{\}, (\Delta\_P \bowtie Q) \cup (P \bowtie \Delta\_Q)>))$
=
/* Transform the second $\cup_\Delta$ using the definition of $\cup_\Delta$ */
$\Delta_+(<P \bowtie Q, \{\}> \cup_\Delta$
$<((\Delta_+P \bowtie Q') \cup (P' \bowtie \Delta_+Q)) - ((\Delta\_P \bowtie Q) \cup (P \bowtie \Delta\_Q)),$
$((\Delta\_P \bowtie Q) \cup (P \bowtie \Delta\_Q)) - ((\Delta_+P \bowtie Q') \cup (P' \bowtie \Delta_+Q))>) =$
/* Transform the remaining $\cup_\Delta$ using the definition of $\cup_\Delta$ */
$\Delta_+<(P \bowtie Q \cup$
$(((\Delta_+P \bowtie Q') \cup (P' \bowtie \Delta_+Q)) - ((\Delta\_P \bowtie Q) \cup (P \bowtie \Delta\_Q))))$
-
$(((\Delta\_P \bowtie Q) \cup (P \bowtie \Delta\_Q)) - ((\Delta_+P \bowtie Q') \cup (P' \bowtie \Delta_+Q))),$
$(((\Delta\_P \bowtie Q) \cup (P \bowtie \Delta\_Q)) - ((\Delta_+P \bowtie Q') \cup (P' \bowtie \Delta_+Q)))$
-
$(P \bowtie Q \cup$
$(((\Delta_+P \bowtie Q') \cup (P' \bowtie \Delta_+Q)) - ((\Delta\_P \bowtie Q) \cup (P \bowtie \Delta\_Q))))>$

1. Note that P and Q represent P_{old} and Q_{old}.

$$
\begin{aligned}
&= \\
&\text{/* Extract the } \Delta_+ \text{ part of the } \Delta\text{-set */} \\
&(P \bowtie Q \cup \\
&(((\Delta_+P \bowtie Q') \cup (P' \bowtie \Delta_+Q)) - ((\Delta\_P \bowtie Q) \cup (P \bowtie \Delta\_Q)))) \\
&- \\
&((\Delta\_P \bowtie Q) \cup (P \bowtie \Delta\_Q)) - ((\Delta_+P \bowtie Q') \cup (P' \bowtie \Delta_+Q)))
\end{aligned}
$$

Note that $P \bowtie Q$ is here directly available since this is the view that is being maintained as a materialized view.

The expression for incrementally maintaining a join above depends on the fact that the future states of sub-expressions are available which is exactly what a breadth-first, bottom-up propagation algorithm guarantees. This expression is what is used for execution in partial differencing where the four partial differentials are kept intact. These are easier to optimize separately than a full differential expression.

Similar, but full, differential expressions can be found in [75]. A major difference is that in partial differencing all net-effects are calculated by the $\cup_\Delta$ operator while in full differential expressions the net-effect calculation is embedded in the full expression. By isolating the net-effect to one operator it is possible to also use partial differencing for propagating physical changes (e.g. physical events of ECA-rules) by using $\cup$ instead of $\cup_\Delta$ (see section 6.16 and section 7.7).

In view maintenance the new value, P', of a view P will usually become the new cached value. If we then want to go back to the previous value of the view P, given P', we can use $P = (P' \cup \Delta\_P) - \Delta_+P$ to perform a *decremental computation* (or a logical rollback). Similarly to the incremental expressions above, we can define the decremental expressions of select (σ), project (π), and join ($\bowtie$) by:

$$\sigma_{cond}P = ((\sigma_{cond}P)' \cup \sigma_{cond}\Delta\_P) - \sigma_{cond}\Delta_+P$$

Note that $(\sigma_{cond}P)'$ is here directly available since this is the view that is being maintained as a materialized view.

$$\pi_{attr}P = ((\pi_{attr}P)' \cup \pi_{attr}\Delta\_P) - \pi_{attr}\Delta_+P$$

Note that $(\pi_{attr}P)'$ is here directly available since this is the view that is being maintained as a materialized view.

$$
\begin{aligned}
P \bowtie Q = ((P \bowtie Q)' \cup \\
((\Delta\_P \bowtie Q) \cup (P \bowtie \Delta\_Q)) - ((\Delta_+P \bowtie Q') \cup (P' \bowtie \Delta_+Q)))) \\
- \\
(((\Delta_+P \bowtie Q') \cup (P' \bowtie \Delta_+Q)) - ((\Delta\_P \bowtie Q) \cup (P \bowtie \Delta\_Q)))
\end{aligned}
$$

Note that $(P \bowtie Q)'$ is here directly available since this is the view that is being maintained as a materialized view.

Here we also depend on the breadth-first, bottom-up propagation algorithm to ensure that old states of the sub-expressions are available when the partial differentials are to be executed.

6.8 The Propagation Algorithm

A breadth-first, bottom-up propagation algorithm has been implemented to support the partial differencing calculus. In the implementation Δ-sets are represented as temporary materializations done in the propagation algorithm and are discarded as the propagation proceeds upwards. Changes, i.e. Δ-sets, which are not referenced by any partial differentials further up in the network are discarded. This assumes that there are no loops in the network, which is not possible with *recursive* relations[1]. The algorithm propagates changes breadth-first by first executing all affected partial differentials of an edge and then by accumulating the changes in the nodes above (fig. 6.9).

Here is an outline of the quite simple algorithm (see chapter 7 for more details):

```
for each level (starting with the lowest level)
    for each changed node (marked as changed when a node below
                           gets a non-empty Δ-set)
        for each edge to a node below
            execute the partial differential(s)
            and accumulate the result in the
            Δ-set of the node using ∪Δ
        if the Δ-set is non-empty
            mark the nodes above as changed
```

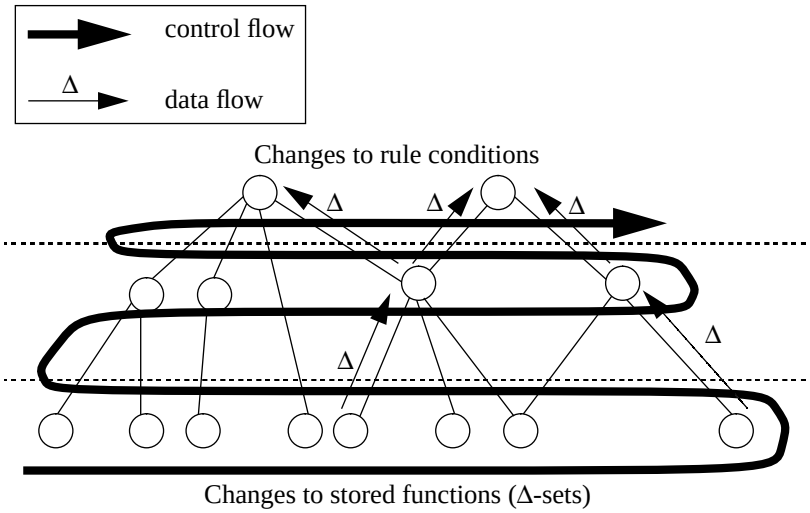


Figure 6.9: Propagation by a breadth-first, bottom-up algorithm

1. The algorithm can be extended to handle linear recursion by revisiting nodes below and using fixed point techniques. Work on incremental evaluation of recursive expressions can be found in [65].

Note that the nodes in the lowest level represent changes to stored functions and are marked as changed during the transaction when the changes are detected and accumulated in the corresponding Δ-set. The Δ-sets of each node are cleared after the node has been processed, i.e. after the partial differentials that reference the Δ-sets have been executed.

In section 7.7 the propagation algorithm is discussed in more detail.

6.9 Performance Measurements

Performance of rule condition monitoring is related to expressibility and the complexity of rule conditions. Expressibility relates to the number of rules needed for a specific monitoring task. The rules in AMOSQL have the full expressibility of AMOSQL queries in the condition. Complexity relates to the number of changes that can affect a rule condition and how they affect the condition. One rule can monitor several different changes to one rule condition. The incremental evaluation technique based on partial differencing is efficient for small changes to a few functions that affect the rule condition, but is not so efficient for large changes to many such functions. The potential number of changes that can affect the rule condition does not directly relate to the efficiency of this technique since it only considers one change at a time. What affects performance is the actual number of updates and how a particular update affects the condition.

A performance measurement was performed using two implementations of rule condition evaluation, one based on naive, i.e. full, evaluation and another based on partial differencing. The benchmarks were based on monitoring the `monitor_items` rule defined previously and with full expansion of rule conditions. Seven benchmarks were run and with encouraging results. The first three measurements aimed at determining how much more efficient the incremental change monitoring is than the naive change monitoring for small changes. These consider few changes per transaction to one, two, or three partial differentials. These are considered normal cases and are shown to be very efficient to monitor using partial differencing. The next three consider many changes to one, two, or three partial differentials. These are considered worst case situations, and are more efficient to monitor naively, but they are still monitored with an acceptable efficiency using partial differencing. The last benchmark modifies the rule condition to contain a function that affects all items with one update; this function will now create one very expensive partial differential. This last benchmark considers one change to this expensive partial differential together with many changes of another cheap partial differential.

6.9.1 Benchmark 1: One Change to One Partial Differential

A series of 100 transactions was run where each transaction changed the quantity of one item. This causes change to only one partial differential in each transaction in the incremental change monitoring. The reason for this can be

seen in fig. 6.4 where changes to quantities (Δ`quantity`) will be propagated by executing only the partial differential Δ`cnd_monitor_items`/Δ_+`quantity`. By contrast, the naive method goes through all the quantities of all the items in the database. The results can be seen in fig. 6.10. Note that the axis have logarithmic scale since the magnitude between the execution times of the different techniques is too great to display with a linear scaled axis. The time for incremental change monitoring is very constant, regardless of the number of items, with an average time of 14 sec. or 140 msec/transaction. Note that the times presented here do not represent the possible throughput of the AMOS architecture, but only the results from running a prototype implementation as a regular application process.[1] The time for naive change monitoring increases linearly with the number of items and is on the average 8.2 sec/transaction for 10 000 items.

From this first benchmark it is easy to see why incremental change monitoring is the better technique of the two when the number of changes to a rule condition in a transaction is small. Naive change monitoring quickly becomes infeasible as the size of the database grows and where rule conditions are complex queries over large portions of the database.

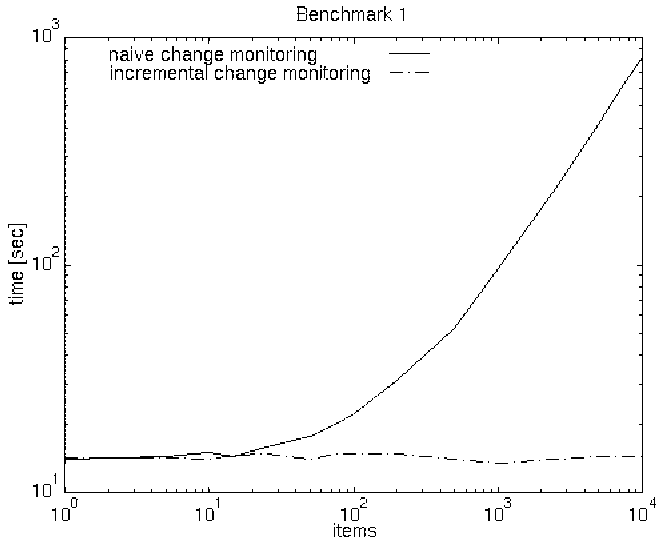


Figure 6.10: 100 transactions with 1 change to 1 partial Δ-relation

6.9.2 Benchmark 2: One Change to Two Partial Differentials

A series of 100 transactions was run where each transaction changed the quan-

1. All measurements were made on an HP9000/710 with 64 Mbyte of main memory and running HP/UX.

tity of one item and the delivery time for the item. This will cause change to two partial differentials which is still very efficient to monitor by incremental techniques. The naive change monitoring technique will evaluate the whole condition and thus increases linearly with the size of the database. The results can be seen in fig. 6.11. The time for incremental change monitoring is on the average 15 sec. or 150 msec/transaction.

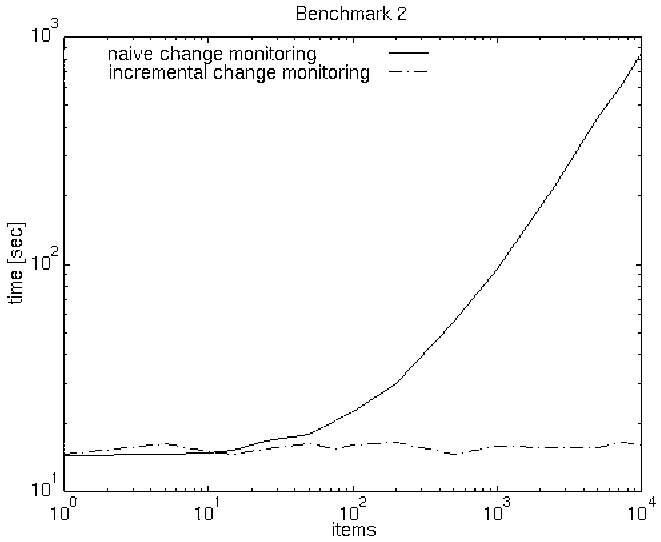


Figure 6.11: 100 transactions with 1 change to 2 partial Δ-relations

6.9.3 Benchmark 3: One Change to Three Partial Differentials

A series of 100 transactions was run where each transaction changed the quantity of one item, the delivery time for the item, and the consume-frequency for the item. This will cause change to three partial differentials which is still very efficient to monitor using incremental techniques. The naive change monitoring technique will evaluate the whole condition and thus increases linearly with the size of the database. The results can be seen in fig. 6.12. The time for incremental change monitoring is on the average 16 sec. or 160 msec/transaction.

These first measurements show that the incremental change monitoring technique is very efficient if the number of changes is small even if several parts of the rule condition are effected. As will be shown in benchmark 7 this is not always the case. It depends on how the change affects the condition and how expensive the related partial differential is to evaluate.

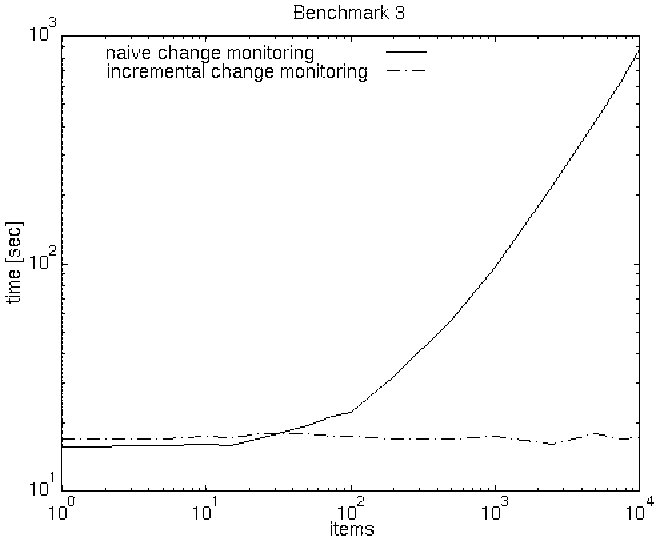


Figure 6.12: 100 transactions with 1 change to 3 partial Δ-relations

6.9.4 Benchmark 4: Many Changes to One Partial Differential

In this test one transaction was run which updated the quantity of all items in the database. This means that only one partial differential (Δcnd_monitor_items/Δ_+quantity) is affected. The affected partial differential has to check the quantities of all the items which is exactly what the naive change monitoring technique does. Since there is an overhead in doing the actual propagation, the incremental change monitoring technique performs slightly worse than the naive one (fig. 6.13).

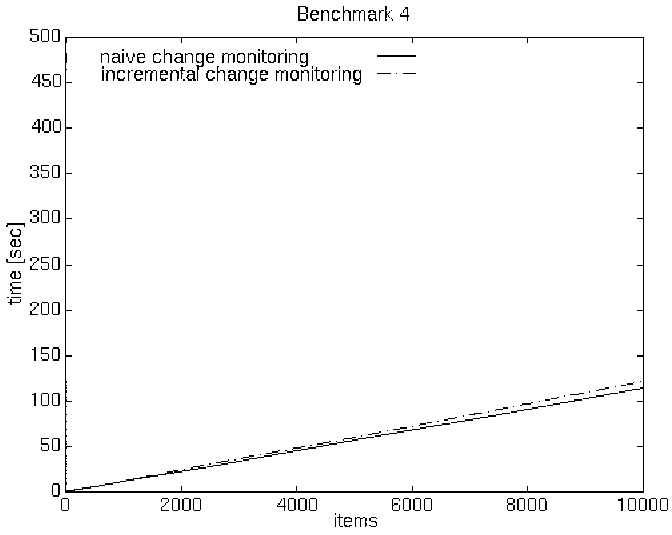


Figure 6.13: 1 transaction with n changes to 1 partial Δ-relation

6.9.5 Benchmark 5: Many Changes to Two Partial Differentials

In this test one transaction was run which updated the quantity and the delivery time of all items in the database. This means that two partial differentials are affected. As shown in fig. 6.4 the partial differentials Δ`cnd_monitor_items/`Δ_+`quantity` and Δ`cnd_monitor_items/`Δ_+`delivery_time` will both need to be executed, which results in overlapping execution. In the naive version these overlaps in the execution do not appear. As shown in fig. 6.14 many changes to three partial differentials perform slightly worse than naive change monitoring. The affected partial differentials have to check all the items which is exactly what the naive change monitoring technique does, but it does it all at once, as in the case of benchmark 4. The incremental change monitoring technique still performs only slightly worse than the naive one.

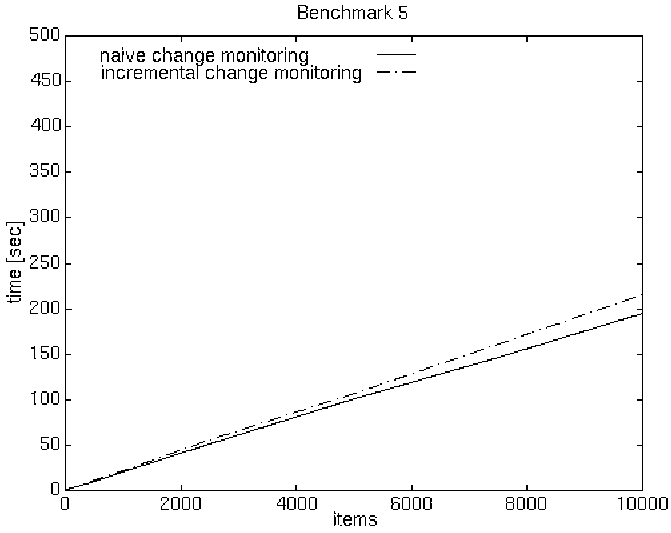


Figure 6.14: 1 transaction with n changes to 2 partial Δ-relations

6.9.6 Benchmark 6: Many Changes to Three Partial Differentials

In this test one transaction was run which updated the quantity, the delivery time, and the consume-frequency of all items in the database. This caused changes to three out of the five partial differentials in each transaction in the incremental change monitoring. As shown in fig. 6.4 the partial differentials Δ`cnd_monitor_items`/Δ_+`quantity`, Δ`cnd_monitor_items`/Δ_+`delivery_time`, and Δ`cnd_monitor_items`/Δ_+`consume_frequency` will all need to be executed, which results in overlapping execution. In the naive version these overlaps in the execution do not appear. As shown in fig. 6.15 many changes to three partial differentials perform worse than naive change monitoring, but only by a constant factor of about 1.6. The affected partial differentials have to check all the items, which is exactly what the naive change monitoring technique does, but it does it all at once, as in the case of benchmarks 4 and 5. The incremental change monitoring technique now performs much worse than the naive one. The reason for this can be found in the definition of the `threshold` function. Changing the quantity or the consume-frequency causes partial differentials to be evaluated that both have to check which supplier supplies the changed item, what the delivery time of the item is, and what the minimum stock of the item is. Changing the delivery time does not require finding the supplier since this is part of the propagated change.

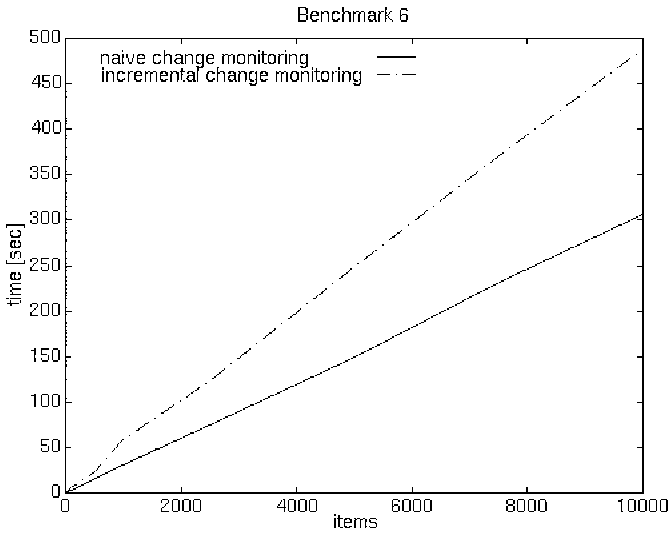


Figure 6.15: 1 transaction with n changes to 3 partial Δ-relations

6.9.7 Benchmark 7: One Change to One Expensive Partial Differential and Many Changes to One Cheap Partial Differential

The previous benchmark shows that different changes can have different effects on performance because of the partial differentials that they affect varies in cost of evaluation. The number of changes of a partial differential does not necessarily need to be large in order to have a large effect on the total performance. To highlight this we redefine the `min_stock` function to affect all items:

```
create function min_stock() -> integer;
create function threshold(item i) -> integer as
     select consume_frequency(i) * delivery_time(i, s)
                         + min_stock()
     for each supplier s where supplies(s) = i;
```

Changing the minimum stock has a dramatic effect on performance. The quantity was changed for all items and the minimum stock was changed once in a single transaction fig. 6.16. This can be compared with fig. 6.13 to show that changing the minimum stock only once dramatically degrades performance for the incremental change monitoring technique.

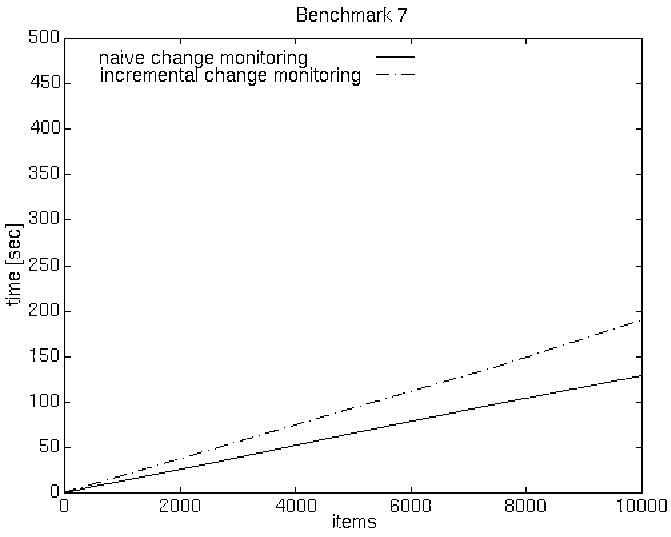


Figure 6.16: 1 transaction with n changes to 1 partial Δ-relation and 1 change to 1 expensive partial Δ-relation

From these measurements the conclusion can be made that the incremental change monitoring technique is superior for a small number of changes in most transactions. In this case, the performance is independent of the size of the database; we say that the incremental change monitoring *scales-up* with respect to size.

For a large number of changes in a transaction the naive change monitoring technique performs better. In the worst case incremental change monitoring, however, only performs worse than naive change monitoring by a constant factor. By deactivating rules with incremental change monitoring during a large number of changes and activating them (causing a naive evaluation) before transactions are committed, the best of both techniques can be attained. However, the cost of deactivating and activating a rule again must be considered here since this involves contracting and then expanding the propagation network.

Note that these measurements are not really dependent on the fact that only one rule is activated. If several rules are activated, but only one of them is affected by the changes, i.e. if the condition refers to the function that changes, then there will be no overhead from the other rules. If, however, there are changes that affect several rules or if one rule causes changes that affect the condition of another rule, then it is a different matter. Measuring performance in such cases requires carefully designed benchmarks that can give valuable information on where the bottlenecks are in different change monitoring and action execution strategies.

6.10 Optimization Techniques for Partial Differencing

There are several optimization techniques that can be used to improve the performance of partial differencing. One basic idea was that it should be possible to optimize the partial differentials with general query optimization techniques. There are also some additional optimization techniques that can be considered for improving the propagation in the propagation network.

6.10.1 General Optimization Techniques

The ObjectLog optimizer in AMOS as described in [82] is based on Horn clause rule substitution and a cost model for subgoal reordering. Horn clause rule substitution means that the optimizer combines Horn clause rules into larger rules by expanding subgoals. Not all subgoals can be expanded, e.g. late bound calls and recursion. By expanding all possible subgoals the following steps in the optimization process will have more degrees of freedom for optimization.

For any Horn Clause rule or predicate P, the input tuple is the tuple corresponding to the variable(s) that are bound in P. For a given input tuple there are zero or several output tuples, corresponding to unbound variable(s) in P. Subgoal reordering is based on a cost model[1] that calculates two cost estimates for P:

1. The *execution cost* of P, C_P, defined as the number of visited tuples, given that all variables of the input tuple are bound.

2. The *fanout*, F_P, which is the estimated number of output tuples produced by P for a given input tuple.

For a conjunctive query consisting of subgoals $\{P_i\}$, $1 \le i \le n$, the total cost C is calculated by the formula:

$$C = \sum_{i=1}^{n} \left(C_{P_i} \prod_{j=1}^{i-1} F_{P_j} \right)$$

For disjunctive queries, i.e. in disjunctive normal form, each part of the disjunction is optimized separately.

A *rank* for each subquery in a query plan is calculated by using fanout and cost information and some optimization strategy. The rank is used to reorder the subgoals in a query plan. In the system three different optimization strategies are available. A heuristic method based on calculating the ranks through a simple formula [82] is currently the default method. A randomized method

1. Assuming that the query execution uses a nested-loop join.

based on Simulated Annealing and Iterative Improvement [73] is available as an option, and which is the most effective of the three for optimizing large queries, i.e. large join queries. Exhaustive optimization is also available [107] which calculates the optimal plan, but can only be used for smaller queries.

The fanout of predicates, i.e. stored relations, is currently defined by the following default values:

- $F_P = 1$ if the input tuple has a unique index.
- $F_P = 2$ if it has a non-unique index.
- $F_P = 4$ otherwise.

The defaults for C_P of predicates are:

- $C_P = F_P$ if the input tuple has an index.
- $C_P = 100$ if it is unindexed, since the system has to scan the entire table.

Foreign predicates (i.e. foreign functions) have by default $F_P = 1$ and $C_P = 1$, assuming they are cheap to execute and return a single result tuple. The user can provide cost hints for each predicate, which override the default assumptions about C_P and F_P. For Horn Clause rules, i.e. for views, F_P is calculated by using F_P of the subgoals.

The reordering of subgoals of a relation P, i.e. a Horn Clause, is performed by the optimizer by using the given C_Q and F_Q of each subgoal Q of P, with the aim of minimizing C_P.

Take an unoptimized version of ObjectLog code for `cnd_monitor_items`:

```
cnd_monitor_items_item,item(I) ←
  quantity_item,integer(I,_G1) ∧
  consume_frequency_item,integer(I,_G2) ∧
  supplies_supplier, item(_G3, I) ∧
  delivery_time_item,supplier,integer(I,_G3,_G4) ∧
  _G5 = _G2 * _G4 ∧
  min_stock_item,integer(I,_G6) ∧
  _G7 = _G5 + _G6 ∧
  _G1 < _G7
```

Here `I` is bound by the scan of `quantity` and since `I` is the unique index of `consume_frequency` it will get $C_{\texttt{consume\_frequency}} = F_{\texttt{consume\_frequency}} = 1$. Since `I` is not an index to `supplies` we get $C_{\texttt{supplies}} = F_{\texttt{supplies}} = 4$. Assuming `I` is not a unique index for `delivery_time`, i.e. different suppliers can be defined to have different delivery times for the same item, we get $C_{\texttt{delivery\_time}} = F_{\texttt{delivery\_time}} = 2$. By swapping `supplies` and `delivery_time` we bind `_G3` by the indexed access of `delivery_time` and

we get $C_{supplies} = F_{supplies} = 1$.

Using the cost formula above the query optimizer will determine that the following query plan is more efficient.

```
cnd_monitor_items_item,item(I) ←
  quantity_item,integer(I,_G1) ∧
  consume_frequency_item,integer(I,_G2) ∧
  delivery_time_item,supplier,integer(I,_G3,_G4) ∧
  supplies_supplier, item(_G3, I) ∧
  _G5 = _G2 * _G4 ∧
  min_stock_item,integer(I,_G6) ∧
  _G7 = _G5 + _G6 ∧
  _G1 < _G7
```

6.10.2 Optimizing Partial Differentials

When optimizing a partial differential the optimizer should take into account that the Δ-relation for which it is differentiated for is much smaller than the original relation. The ObjectLog optimizer described in [82] is being extended with new cost metrics for Δ-relations. Take the unoptimized version of Δcnd_monitor_items/Δ_+supplies:

```
Δcnd_monitor_items_item(I)/Δ+supplies ←
  quantity_item,integer(I,_G1) ∧
  consume_frequency_item,integer(I,_G2) ∧
  delivery_time_item,supplier,integer(I,_G3,_G4) ∧
  Δ+supplies_supplier, item(_G3, I) ∧
  _G5 = _G2 * _G4 ∧
  min_stock_item,integer(I,_G6) ∧
  _G7 = _G5 + _G6 ∧
  _G1 < _G7
```

By defining fanout and cost for Δ-relations that give a much cheaper total cost the query optimizer moves the Δ-relations early in a query plan (usually first).

For example, we can define the fanout and cost defaults for a predicate P, ΔP by:

- $F_{\Delta P} = 0.1$ if the input tuple has a unique index.
- $F_{\Delta P} = 0.2$ if it has a non-unique index.
- $F_{\Delta P} = 0.4$ otherwise

with the defaults for $C_{\Delta P}$ defined by:

- $C_{\Delta P} = F_P$ if the input tuple has an index.

- $C_{\Delta P} = 2$ if it is unindexed, since the system has to scan the entire event history.

From these we get $C_{\Delta supplies} = F_{\Delta supplies} = 0.4$ and the query optimizer will find that the following query plan is more efficient:

```
Δcnd_monitor_items_item(I)/Δ+supplies ←
  Δ+supplies_supplier, item(_G3, I) ∧
  quantity_item,integer(I,_G1) ∧
  consume_frequency_item,integer(I,_G2) ∧
  delivery_time_item,supplier,integer(I,_G3,_G4) ∧
  _G5 = _G2 * _G4 ∧
  min_stock_item,integer(I,_G6) ∧
  _G7 = _G5 + _G6 ∧
  _G1 < _G7
```

Since Δ-relations are usually very small they will often be moved early. Often the Δ-relation will be placed first, but this is not always the case as is assumed in [65]. For example, if some foreign predicate is defined to be very cheap, it might produce a more efficient plan if it is placed before a Δ-relation. A Δ-relation could also be defined to be more expensive than indexing a stored relation, e.g. if we assume that it represents a function that will be changed quite heavily during a transaction.

The fanout and cost of a differential can be application dependent, especially if it represents changes to external data. An application can update some data very often while other data is changed more seldom. The AMOS query optimizer allows the user to specify fanout and cost for a function and in section 9.5.2 this is discussed as part of the definition of foreign data sources that defines external data in a database schema. The actual choice of default values for fanout and cost will have to be investigated further. The above default values have been chosen to produce a lower cost for query plans with partial differentials early, but are not backed up by any empirical studies or benchmarks.

In [75] a framework for query optimization of differentiated queries is presented. Some of these rules such as performing selections or projections to smaller differentials instead of full expressions are covered by the query optimizations techniques discussed above. In AMOS a differential is given a smaller fanout (and cost) than the full corresponding expression causing the query optimizer to choose selections or projections to differentials before most full expressions.

6.10.3 Common Subqueries and Node Sharing

Optimizations such as reusing sub-expressions can be made by restricting the way AMOSQL functions are expanded when being compiled into ObjectLog. There is a trade-off between expansion for better query optimization and node sharing for more efficient change propagation. This is an area for further research.

To achieve a propagation network analogous to that in fig. 6.3 we could choose to define `cnd_monitor_items` in terms of two partial differentials instead:

$$\Delta\texttt{cnd\_monitor\_items}_{\texttt{item}}(\texttt{I})/\Delta_{+}\texttt{quantity} \leftarrow$$
$$\Delta_{+}\texttt{quantity}_{\texttt{item,integer}}(\texttt{I, \_G1}) \wedge$$
$$\texttt{threshold}_{\texttt{item,integer}}(\texttt{I, \_G2}) \wedge$$
$$\texttt{\_G1} < \texttt{\_G2}$$

$$\Delta\texttt{cnd\_monitor\_items}_{\texttt{item}}(\texttt{I})/\Delta_{+}\texttt{threshold} \leftarrow$$
$$\texttt{quantity}_{\texttt{item,integer}}(\texttt{I, \_G1}) \wedge$$
$$\Delta_{+}\texttt{threshold}_{\texttt{item,integer}}(\texttt{I, \_G2}) \wedge$$
$$\texttt{\_G1} < \texttt{\_G2}$$

The Δ`threshold` function would then be defined in terms of four partial differentials and become an intermediate node in the network. This would be beneficial if the threshold function is referenced in other rule conditions as well since this would enable node sharing.

6.11 Strict and Nervous Rule Semantics

Partial differentials that contain selections might produce Δ-sets that are too large. This occurs, for example, if we want to calculate $\Delta\sigma_{\text{employee.income>10000}}(\text{employee})$ where we get an update of employee.income from 10 001 to 10 002. Such an update causes no change to the set of all employees with a salary > 10 000 while a partial differential of such an expression will report an update. This is acceptable for rule conditions only dependent on positive changes and that use nervous semantics.

For strict semantics these tuples have to be removed by checking the old state of the selection. In the example above we have to materialize the set of all employees with a salary > 10 000. Negative partial differentials might also produce a Δ-set that is too large, i.e. deletions of tuples that are still present in the new state of the database. Unlike for positive changes, this is more serious as it might cause rules not to trigger on positive changes since these have been cancelled by incorrectly propagated negative changes. To avoid this, for negative changes we have to check if the tuple is still present in the new state of the database. If this is not done, the rules might under-react, which is unacceptable. For strict semantics of unions a check is made that positive/negative changes are propagated only if the other part of the union was/is not present.

6.12 Bag-oriented and Set-oriented Semantics

Note that we assume *set-oriented semantics* since this is the most natural semantics for rule conditions. Partial differencing can be defined for *bag-oriented semantics* as well, which is discussed in section 6.17. Partial differencing of the relational operators for bag-oriented semantics is not as straight forward as for set-oriented semantics. Some work on differencing where bag-oriented

semantics is assumed can be found in [61][77][100].

In [77] a technique similar to partial differencing is used, but with support for removing overlaps between different partial differentials to have true bag-oriented semantics.

In [77] positive changes are calculated by: changing all subgoals Y in $\Delta P/\Delta_+ X$ to $Y - \Delta_+ Y$, $\forall X, Y \in D_p$ and $X \neq Y$ and where Y precedes $\Delta_+ X$ in the conjunction, and negative changes by: changing all Y_{old} in $\Delta P/\Delta_- X$ to $Y_{old} - \Delta_- Y$, $\forall X, Y \in D_p$ and $X \neq Y$ and where Y_{old} precedes $\Delta_- X$ in the conjunction. (See appendix for how this modification affects partial differencing of conjunctions).

With set-oriented semantics, when there are changes to more than one influent, the definitions in fig. 6.1 might lead to a set of changes that is too large, i.e. containing duplicates. These will, however, be removed by $\cup_\Delta$. Since $\cup_\Delta$ is not commutative for set-oriented semantics, $\cup_\Delta$ has to be performed in the same order as the changes originally occurred in the transaction. For example, if a tuple is first inserted and is later removed, then there is no net effect. If the order of operations is reversed, i.e. a removal (without any effect) followed by an addition, then the net effect is one added tuple. In section 6.17 partial differencing and the $\cup_\Delta$ are extended to handle updates separately. Part of this extension is to define Δ-sets as transaction time relations, i.e. as timestamp ordered time series. The new version of $\cup_\Delta$ joins the Δ-sets in timestamp order and thus can support both set-oriented semantics and bag-oriented semantics[1].

6.13 Partial Differencing of Overloaded Functions

In the previous sections it has not been explained how partial differencing is performed on overloaded functions. If a rule condition references an overloaded function, all the possible resolvents have to be considered when the condition is monitored. In the change propagation of partial differencing the propagation network will have different propagation paths for each resolvent. The network now becomes a graph instead of a a tree structure. Lets look at the `no_high` rule again:

```
create rule no_high(department d) as
    when for each employee e
    where dept(e) = d and
       employee.netincome->number(e) > netincome(mgr(e))
    do set employee.grossincome->number(e) =
                                      grossincome(mgr(e));
```

The condition function for this rule will become:

1. Time series can be used to represent a bag since each duplicate tuple will have a unique timestamp which represents the transaction time when the tuple was inserted into the bag.

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

The query compiler/optimizer can here deduce that there are two different `netincome` functions (resolvents) involved, one for employees and one for managers. There is no confusion of which resolvent to choose, i.e. the compiler will perform early binding. The propagation network for this function can be seen in fig. 6.17.

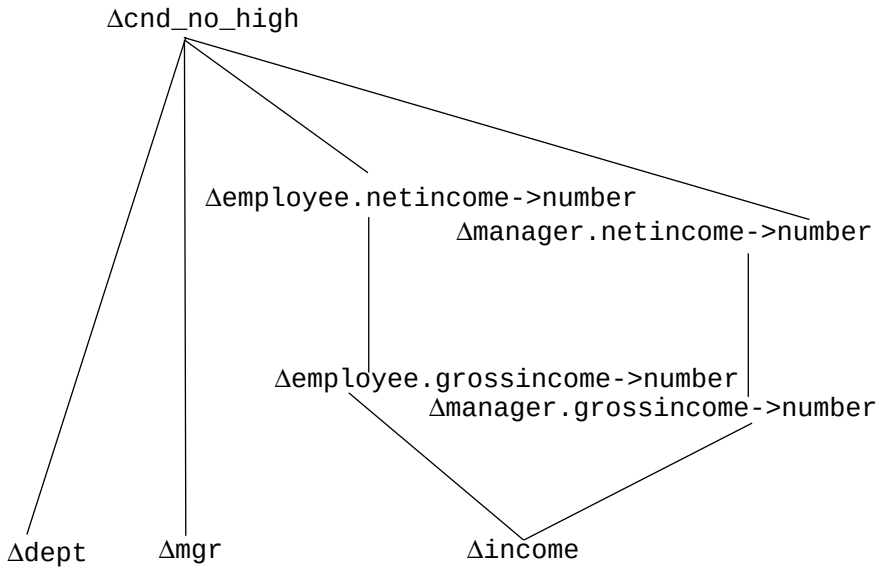


Figure 6.17: The propagation network for `cnd_no_high`

Since both `netincome` and `grossincome` are overloaded, different partial Δ-relations will be defined that can calculate the changes to the full functions in terms of changes to the `income` function. Since the AMOSQL compiler can determine exactly what two different resolvents of `netincome` and `grossincome` are involved here, the final execution plan for the partial differentials will reference these. Actually in the current implementation the execution plan will be expanded to reference the `income` function directly twice, i.e. once for each resolvent. Two different partial differentials will be generated for monitoring `Δincome`, i.e. `Δcnd_no_high/Δincome'` and `Δcnd_no_high/Δincome''`. If this expansion was not done, the propagation network would look as above and we get the partial differentials `Δcnd_no_high/Δemployee.netincome->number` and `Δcnd_no_high/Δmanager.netincome->number`. This can be beneficial if several rules reference these resolvents in their conditions, since it would promote node sharing in the net-

work.

If the compiler cannot determine the correct resolvents at compile-time this will have to done at run-time, i.e. late binding of functions will be done. This would be the case in the rule:

```
create rule no_high(department d) as
    for each employee e
    when dept(e) = d and netincome(e) > netincome(mgr(e))
    do set employee.grossincome->number(e) =
                                        grossincome(mgr(e));
```

The condition function for this rule will become:

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

Here the first reference to `netincome` in the condition can result in either of the two resolvents depending on whether `e` is a manager or not. This will result in a special algebra operator in the execution plan called a DTR (*Dynamic Type Resolver*) [48]. This operator makes access to several resolvents possible and also supports optimization and inverted calls. In a propagation network the DTR will cause the creation of a special node[1] that collects changes from all the possible resolvents for that specific DTR. Each resolvent will then have its own Δ-set with different changes. Which changes (which Δ-set) that should be propagated is determined at run-time. This node can be seen named Δ`netincome`$_{\mathtt{DTR}}$ in the propagation network in fig. 6.18.

6.14 ECA-rule Semantics

Take an ECA-rule r() defined as:

for each ‘type of x’ x, ‘type of y’ y
on e1(x) or e2(x)
when c1(x) and c2(y)
do a1(x) and a2(y).

Using the notations defined in section 6.5 we can define an ECA-rule as:

<name>(<parameter-specification>) =
[<variable quantification>]
<event-specification> | (<condition> ⇒ <action>)

We can now write the rule r as:

1. Not yet implemented.

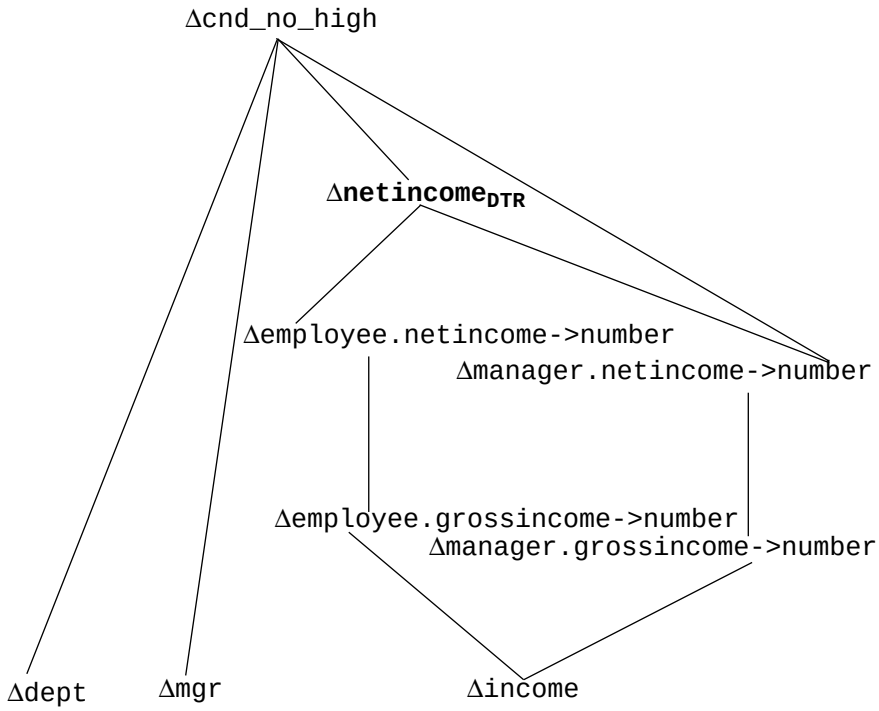


Figure 6.18: The propagation network for `cnd_no_high` with late binding

$$r() = \forall\, x_{\text{type of x}}, y_{\text{type of y}}\ e1(x) \vee e2(x) |\ (c1(x) \wedge c2(y) \Rightarrow a1(x) \wedge a2(y)),$$

The event part is monitored by defining a derived *event function* f_e that accesses other event functions and that returns the data shared by the event and the condition (here data of the type of x):

$$f_e()_{\text{type of x}} = \forall\, x_{\text{type of x}}\ \text{select x where } e1(x) \vee e2(x),$$

As before we define a *condition function* f_c that returns the data shared by the condition and action (here data of the type of x and of the type y):

$$f_c(x_{\text{type of x}})_{\text{type of x, type of y}} = \forall\, y_{\text{type of y}}\ \text{select x, y where } c1(x) \wedge c2(y),$$

i.e. a function that returns a set of values of type x for all c(x) that return true, and an *action procedure* f_a that takes the output of the condition function (here the of type of x and type of y) as argument,

$$f_a(x_{\text{type of x}}, y_{\text{type of y}}) = a1(x) \wedge a2(y)$$

ECA-rule execution can now be seen as the function application $f_a(f_c(f_e()))$ where f_e only returns a value if the whole event specification is true. More pre-

cisely for nervous and strict rule execution, respectively, we define:

$\forall\ x_{\text{type of x}}, y_{\text{type of y}}$
where $x \in f_e(x) \wedge \langle x, y\rangle \in \Delta f_c(x)$[1]
do $f_a(x, y)$

$\forall\ x_{\text{type of x}}, y_{\text{type of y}}$
where $x \in f_e(x) \wedge \langle x, y\rangle \in \Delta f_c(x) \wedge \langle x, y\rangle \notin (f_c)_{old}(x)$
do $f_a(x, y)$

Note that here x is bound from the event part, but y is free and is fetched from the database. In AMOS the action function and the condition function are executed at the same time, i.e. in the propagation phase. Conflict resolution is, of course, applied to a triggered ECA-rule before the action function is actually executed.

For EA-rules the condition function f_c can be skipped in the definitions above and the result from the event function f_e is directly passed to the action function f_a. Of course the separation between nervous and strict rule execution semantics does not apply to EA-rules since it only applies to rules with logical conditions and EA-rules only monitor physical events.

In contrast, CA-rules only monitor logical events and ECA-rules monitor both physical and logical events.

6.15 Propagating Events of ECA-rules

Propagating events in AMOS [86] is very similar to propagation of changes to conditions. One major difference is that here we distinguish between three basic changes (events) on stored functions. Δ_+ and Δ_- represent insertions and removals to/from a stored function (using the AMOSQL `add` and `remove` operations). Updates (AMOSQL `set` operation) are handled separately using the Δ_{-+}-function. In partial differencing updates are defined as one or several Δ_- events immediately followed by a Δ_+ event. Partial differencing can be defined in terms of updates, i.e. Δ_{-+}-functions, as well (see section 6.17). The Δ_{-+}-function is defined to return the new tuple from an update along with a bag of the old tuples that were removed as a consequence of the update[2]. The Δ-functions represent how event histories are defined in AMOS (see chapter 8). The Δ-functions (or Δ-relations) of stored functions are called *stored event functions* and *derived event functions* for derived functions. In an ECA-rule such as:

1. Actually we execute the action procedure f_a on the positive changes representing additions ($\Delta_+ f_c$) and updates ($\Delta_{-+} f_c$) of f_c.
2. In the general case an update should be defined as a new bag and an old bag, but here only tuple-based updates are assumed to simplify the discussion.

```
create rule check_new(department d) as
    for each employee e, manager m
    on updated(dept(e)) and updated(income(e))
    when dept(e) = d and
         m = mgr(e) and
         employee.netincome->number(e) > netincome(m)
    do rollback;
```

the events that trigger the rule have to be calculated in a similar manner as in partial differencing. During the compilation of ECA-rules a derived event function is created (along with the condition and action functions). For the `check_new` rule these functions would be:

```
create function evt_check_new(department d_new)
                        -> <department d_new, employee e>
    as select d_new, e
    for each bag of department d_old,
             bag of number n_old, number n_new
    where <d_old, d_new> = Δ-+dept(e) and
          <n_old, n_new> = Δ-+income(e);

create function cnd_check_new(department d, employee e)
                                    -> boolean as
    select true for each manager m
    where dept(e) = d and
         m = mgr(e) and
         employee.netincome->number(e) > netincome(m);

create function act_check_new(boolean)-> boolean as
         rollback;
```

The event function (or Δ-functions) can be referenced in the conditions of ECA-rules as ordinary functions by passing the event data from the event function to the condition function, e.g. fetching the previous value of an update. When the rules are checked the event functions are evaluated and if they return anything, i.e. if the rule has been triggered, the result can be passed to the condition and action functions. This gives a limited kind of incremental evaluation on functions in the condition and action that share variables with the event part. A triggering graph (fig. 6.19) is used to determine which event functions should be executed.

In this ECA-rule the value n_{new} returned by the stored event function Δ_{-+}`income(e)` is not used for calculating `netincome` in the condition. The variable `e` is shared between the event and the condition makes the calculation of the `netincome` function efficient since `e` is the primary key to the updated `income` function which is accessed by `netincome`. In most cases this provides efficient condition evaluation since it is natural to share variables between the event and the condition. In those cases where the condition refer-

ences a complex view (derived function) that does not share any variables with the event part, the technique is modified to use partial differencing for change monitoring of the rule conditions as well as for event propagation.

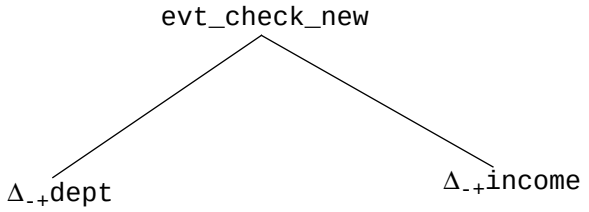


Figure 6.19: The triggering graph for `evt_check_new`

6.16 Event Propagation vs. Partial Differencing of Rule Conditions

Event propagation and rule condition change monitoring can be defined to coexist. The major difference is that event propagation is based on physical changes and change propagation is based on logical changes. By adding information to network nodes if they are part of event propagation, the physical changes can be kept during the propagation. Note that physical changes will always be translated into logical changes if they are to be used in condition change monitoring. The main requirement is that we have to keep all physical changes if they are needed as physical events in an event expression further up in the network.

Note that physical changes are not only limited to stored relations. If an ECA-rule references a view in the event part, then we must keep all physical changes to that view.

In section 6.17 partial differencing is extended to handle updates. The Δ-set is extended to contain a Δ_{-+} part and the $\cup_{\Delta}$ operator is extended to support the union of the extended Δ-sets. By using the $\cup$ operator instead of $\cup_{\Delta}$ all physical events calculated by partial differentials will be propagated. Partial differencing of intersections (conjunctions), unions (disjunctions), and complements (negations) are also modified to handle the updates. ECA-rules in AMOS [86] that reference complex events, i.e. changes to derived functions, use a similar propagation network as is used for partial differencing. This technique is limited to derived functions that are directly dependent on the updates of the stored functions that they reference, i.e. they cannot contain any selections. Event propagation of more complex derived functions can use partial differencing, but with propagation physical changes instead of logical ones. This has not been implemented yet, but is part of ongoing and future work. In chapter 9 monitoring of foreign functions is also considered as a further extension that

can use the change and event propagation techniques discussed here.

Let us look at an alternative version of the check_new rule with an event part referencing a derived function:

```
create rule check_new(department d) as
    for each employee e, manager m
    on updated(dept(e)) and
       updated(employee.netincome->number(e))
    when dept(e) = d and
         m = mgr(e) and
         employee.netincome->number(e) > netincome(m)
    do rollback;
```

Now the generated functions will be:

```
create function evt_check_new(department d_new)
                        -> <department d_new, employee e>
    as select d_new, e
    for each department d_old,
             number n_old,number n_new
    where <d_old, d_new> = Δ-+dept(e) and
          <n_old, n_new> = Δ-+netincome(e);

create function cnd_check_new(department d, employee e)
                                    -> boolean as
    select true for each manager m
    where dept(e) = d and
         m = mgr(e) and
         employee.netincome->number(e) > netincome(m);

create function act_check_new(boolean)->boolean as
         rollback;
```

The Δ_{-+}netincome cannot be calculated directly since it is a derived function. Instead we must use an event propagation network (fig. 6.20) which is a generalization of the triggering graph and which is very similar to the propagation network in partial differencing.

The calculation of Δnetincome will be done using partial differencing, but using physical changes instead of logical ones. Event functions like evt_check_new are calculated directly using the propagated changes. The evt_check_new will directly access the events in Δ_{-+}dept and those propagated to Δ_{-+}netincome. If several ECA-rules are defined, the propagation network will propagate all the changes breadth-first, bottom-up in a similar way as for propagation of changes to rule conditions. The event propagation network propagates all the physical changes and the event functions will be able to access all the changes to the stored and derived functions that they reference. Techniques for optimization of partial differentials and network node

sharing are analogous to those for condition change monitoring.

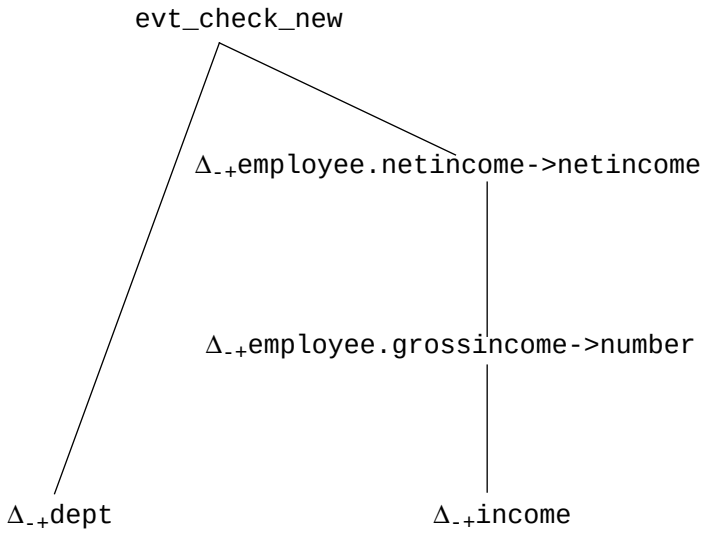


Figure 6.20: The event propagation network for `evt_check_new`

By combining this technique with that of change propagation for partial differencing, event propagation networks and change propagation networks can share nodes. Note that incremental evaluation of conditions in ECA-rules is usually only needed for functions that do not use shared variables passed between the event and the condition. If a function call involves a shared variable, the propagated event data is passed from the event part to the condition in a kind of user-defined incremental evaluation.

One major difference between propagating events to event functions and propagating changes to incrementally calculate condition functions is that in the first case physical events are propagated and in the other case logical events (the logical changes) are propagated where conflicting events are cancelled out by the $\cup_\Delta$. The physical and logical propagation can be viewed as two separate sub-networks of that can share nodes (fig. 6.21).

,

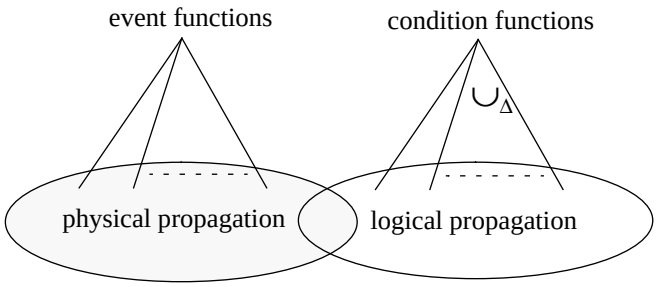


Figure 6.21: Propagating physical events to event functions and logical events (the logical changes) to condition functions

If physical events are propagated to the logical propagation network they will be transformed into logical events by the $\cup_\Delta$ only when they are not needed as physical events in higher levels of the propagation network. The combined network for propagating events to `evt_check_new` and changes to Δ`cnd_check_new` can be seen in fig. 6.22.

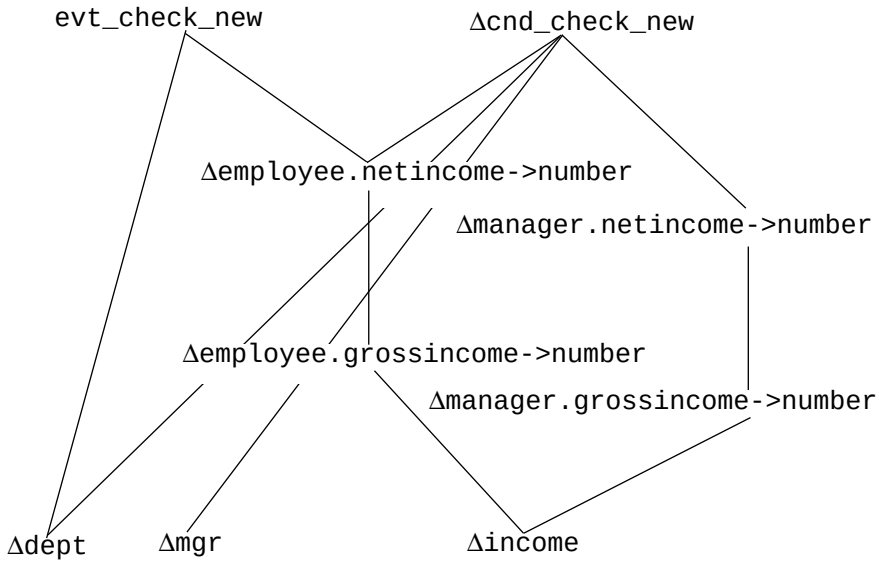


Figure 6.22: The combined propagation network for `evt_check_new` and Δ`cnd_check_new`

Here changes to Δ`employee.netincome->number` will be kept as physical events since they are needed by `evt_check_new`. They will be transformed to logical events when they are added to Δ`cnd_check_new` by $\cup_\Delta$.

6.17 Extended Partial Differencing Calculus for Updates

The added and removed of tuples of a relation P are defined as $\Delta_+P(t_a)$ and $\Delta_-P(t_r)$ where t_a and t_r are the transaction times of the add and remove operations, respectively. The update of a relation P is defined as $\Delta_{-+}P = \{<\{-<P, t_u, key_n, a_1, ..., a_k>\}, \{+<P, t_u, key_n, b_1, ..., b_k>\}>_n\}$, where $\{+<P, t_u, key_n, b_1, ..., b_k>\}$ is a new added bag of tuples, t_u is the time of the update, and k is the arity of P. $\{-<P, t_u, key_n, a_1, ..., a_k>\}$ is the bag of tuples that the update removed and $n \geq 0$ is the number of updates to P.

The definition of a Δ-set is now modified to:

$\Delta P = <\Delta_+P,\Delta_-P,\Delta_{-+}P>$

The *delta-union*, $\cup_\Delta$, over differentials is defined as:

$$
\begin{aligned}
&\Delta P_1 \cup_\Delta \Delta P_2 = \\
&\quad <(\Delta_+P_1(t_a) - \Delta_-P_2(t_r) - \{-<P_2, t_u, key_n, c_1, ..., c_k>\}_n) \text{ where } (t_a<t_r, t_a<t_u) \\
&\qquad \cup \\
&\quad (\Delta_+P_2(t_a) - \Delta_-P_1(t_r) - \{-<P_1, t_u, key_m, a_1, ..., a_k>\}_m) \text{ where } (t_a<t_r, t_a<t_u), \\
&\quad (\Delta_-P_1(t_r) - \Delta_+P_2(t_a) - \{+<P_2, t_u, key_n, d_1, ..., d_k>\}_n) \text{ where } (t_r<t_a, t_r<t_u) \\
&\qquad \cup \\
&\quad (\Delta_-P_2(t_r) - \Delta_+P_1(t_a) - \{+<P_1, t_u, key_m, b_1, ..., b_k>\}_m) \text{ where } (t_r<t_a, t_r<t_u), \\
&\quad \{<\{-<P_1, t_u, key_m, a_1, ..., a_k>\} - \Delta_+P_2(t_a), \text{ where } t_u<t_a \\
&\qquad \{+<P_1, t_u, key_m, b_1, ..., b_k>\} - \Delta_-P_2(t_r) >_m\} \text{ where } t_u<t_r \\
&\qquad \cup \\
&\quad \{<\{-<P_2, t_u, key_n, c_1, ..., c_k>\} - \Delta_+P_1(t_a), \text{ where } t_u<t_a \\
&\qquad \{+<P_2, t_u, key_n, d_1, ..., d_k>\} - \Delta_-P_1(t_r)>_n\}>, \text{ where } t_u<t_r
\end{aligned}
$$

where k is the arity of P, $m \geq 0$ is the number of updates in P_1 and $n \geq 0$ is the number of updates in P_2. Note that the order of the operations is captured by comparing the time stamps. The Δ-sets are stored as tuples of time series (se section 8.6) and will thus behave as bags since duplicates will be stored with different timestamps. The – and ∪ operations above use *bag-oriented semantics*, – removes the latest matching tuples and ∪ joins time series by inserting the tuples into new ordered time series. The $\cup_\Delta$ is now completely bag-oriented and the Δ-sets are really Δ-bags.

Next we modify the definition of the *partial differential*, $\Delta P/\Delta X$, that incrementally monitors changes to P from changes of each influent X. *Partial differencing* of a relation is defined as generating partial differentials for all the influents of the relation. The net changes of the partial differentials are accumulated (using $\cup_\Delta$) into ΔP. The new version of $\cup_\Delta$ above sorts all the added, remove, and updated tuples into the respective parts in the resulting Δ-set.

Let I_p be the set of all relations that P depends on. The Δ-set of P, ΔP, is then defined by:

$$\Delta P = \cup_\Delta \frac{\Delta P}{\Delta X} \overset{1}{=} \cup_\Delta <\frac{\Delta P}{\Delta_+X}, \frac{\Delta P}{\Delta_-X}, \frac{\Delta P}{\Delta_{-+}X}>, \forall X \in I_p$$

For example, if P depends on the relations Q and R then:

1. Equivalent to $\cup_\Delta \Delta P/\Delta X$.

$$\Delta P = \frac{\Delta P}{\Delta Q} \cup_\Delta \frac{\Delta P}{\Delta R} = <\frac{\Delta P}{\Delta_+ Q}, \frac{\Delta P}{\Delta_- Q}, \frac{\Delta P}{\Delta_{-+} Q}> \cup_\Delta <\frac{\Delta P}{\Delta_+ R}, \frac{\Delta P}{\Delta_- R}, \frac{\Delta P}{\Delta_{-+} R}>$$

To detect changes of derived relations we define intersection (conjunction), union (disjunction), and complement (negation) in terms of their differentials as:

$$\begin{aligned}
\Delta(Q \cap R) = {} & <(\Delta_+ Q(t_a) \cap R) \cup (Q \cap \Delta_+ R(t_a)), \{\}, \{\}> \\
& \cup_\Delta \\
& <\{\}, (\Delta_- Q(t_r) \cap R_{old}) \cup (Q_{old} \cap \Delta_- R(t_r), \{\}> \\
& \cup_\Delta \\
& <\{\}, \{\}, \{<\{-<Q, t_u, key_m, a_1, ..., a_j>\}_m \cap R_{old}, \\
& \qquad \{+<Q, t_u, key_m, b_1, ..., b_j>\}_m \cap R>\} \\
& \qquad \cup \\
& \qquad \{<Q_{old} \cap \{-<R, t_u, key_n, c_1, ..., c_k>\}_n, \\
& \qquad\quad Q \cap \{+<R, t_u, key_n, d_1, ..., d_k>\}_n>\}>
\end{aligned}$$

$$\begin{aligned}
\Delta(Q \cup R) = {} & <(\Delta_+ Q(t_a) - R_{old}) \cup (\Delta_+ R(t_a) - Q_{old}), \{\}, \{\}> \\
& \cup_\Delta \\
& <\{\}, (\Delta_- Q(t_r) - R) \cup (\Delta_- R(t_r) - Q), \{\}> \\
& \cup_\Delta \\
& <\{\}, \{\}, \{<\{-<Q, t_u, key_m, a_1, ..., a_j>\}_m - R, \\
& \qquad \{+<Q, t_u, key_m, b_1, ..., b_j>\}_m - R_{old}>\} \\
& \qquad \cup \\
& \qquad \{<\{-<R, t_u, key_n, c_1, ..., c_k>\}_n - Q, \\
& \qquad\quad \{+<R, t_u, key_n, d_1, ..., d_k>\}_n - Q_{old}>\}
\end{aligned}$$

$$\begin{aligned}
\Delta(\sim Q) = {} & <\Delta_- Q(t_r), \Delta_+ Q(t_a), \\
& \qquad \{<\{+<Q, t_u, key_1, b_1, ..., b_j>\}, \{-<Q, t_u, key_1, a_1, ..., a_j>\}>_1\} \\
& \qquad\qquad \cup \\
& \qquad\qquad . \\
& \qquad\qquad . \\
& \qquad\qquad \cup \\
& \qquad \{<\{+<Q, t_u, key_m, b_1, ..., b_j>\}, \{-<Q, t_u, key_m, a_1, ..., a_j>\}>_m\}>
\end{aligned}$$

where j is the arity of Q, k is the arity of R, $m \geq 0$ is the number of updates in Q, and $n \geq 0$ is the number of updates in R. Note that timestamps from each partial differential are propagated and sorted (by $\cup_\Delta$) in timestamp order separately for the added(t_a), removed(t_r), and updated(t_u) parts in the Δ-set that is the result of each of the above expressions. Note that overlaps between different partial differentials still have to be removed to have true bag-oriented semantics as is explained in section 6.12.

Note also that the logical rollback is here defined in terms of updates as well, i.e. $R_{old} = (R \cup \Delta_- R \cup \{-<R, t_u, key_n, c_1, ..., c_k>\}_n) - \Delta_+ R - \{+<R, t_u, key_n, d_1, ..., d_k>\}_n$ and likewise for Q_{old}.

6.18 Partial Differencing of Aggregates

Aggregate functions such as `sum` or `count` must be handled specially when using partial differencing to calculate changes to rule conditions.

Take a rule that limits the number of allowed employees in a department:

```
create rule max_employees(department d,
                          integer max_nr_emp) as
    when count(select e for each employee e
                  where dept(e) = d) > max_nr_emp
    do rollback;
```

The condition function for this rule will be:

```
create function cnd_max_employees(department d,
                          integer max_nr_emp)
    -> boolean as
    select true
        where count(select e for each employee e
                        where dept(e) = d) > max_nr_emp;
```

Bags created in sub-selects in AMOS are created by a special make-bag function that the function compiler creates automatically for each AMOSQL expression that generates a bag. Bags are generated by sub-select expressions and at calls to aggregate functions. The make-bag function takes all the free variables in the sub-select as arguments and returns a generated bag. For the sub-select above the following function is created and is called in the condition function:

```
create function make_bag_cnd_max_employees(department d)
    -> bag of employee as
        select e for each employee e
                        where dept(e) = d;

create function cnd_max_employees(department d,
                          integer max_nr_emp)
    -> boolean as
    select true
     where count(make_bag_cnd_max_employees(d)) > max_nr_emp;
```

By incrementally calculating the value of `make_bag`$_{\texttt{cnd\_max\_employees}}$, using partial differencing[1], we can incrementally maintain the value of the count aggregate without materializing the whole bag.

1. Note that to support correct incremental evaluation of aggregates, bag-oriented semantics of the partial differencing calculus has to be used.

Changes to cnd_max_employees are calculated from changes $\texttt{make\_bag}_{\texttt{cnd\_max\_employees}}$ which in turn is calculated from changes to dept. Two different partial differentials of $\Delta\texttt{make\_bag}_{\texttt{cnd\_max\_employees}}$, called $\Delta\texttt{make\_bag}_{\texttt{cnd\_max\_employees}}/\Delta_{+}\texttt{dept}$ and $\Delta\texttt{make\_bag}_{\texttt{cnd\_max\_employees}}/\Delta_{-}\texttt{dept}$, will generate bags for the added and removed tuples to the bag, respectively. The propagation network for cnd_max_employees can be seen in fig. 6.23.

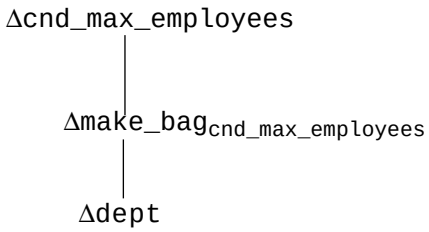


Figure 6.23: The propagation network for cnd_max_employees

To support incremental calculation of the count aggregate the condition function will cache the old value and only call a special incremental version of count with the new changes to the bag:

```
create function cnd_max_employees(department d,
                            integer max_nr_emp)
    -> boolean as
    select true
     where count_cnd_max_employees
                        (Δ+make_bag_cnd_max_employees(d),
                         Δ-make_bag_cnd_max_employees(d), d)
                                  > max_nr_emp;
```

where the incremental aggregate function is defined as:

```
create function count_cnd_max_employees
    (bag of employee ae, bag of employee re, department d)
    -> integer as
    begin
       set cached_count_cnd_max_employees(d) =
              cached_count_cnd_max_employees(d) +
              count(ae) - count(re);
       result cached_count_cnd_max_employees(d);
    end;
```

and where the cache is a stored function:

```
create function cached_count_cnd_max_employees(department d)
```

```
    -> integer;
```

Note that for maintaining the cached aggregate correctly the initial bag has to be calculated completely when the corresponding rule is activated. After the initial value of the aggregate has been determined it can be incrementally maintained. For the above rule the following expression will be evaluated when the rule is activated:

$$\texttt{set cached\_count}_{\texttt{cnd\_max\_employees}}\texttt{(d) =}$$
$$\texttt{count(make\_bag}_{\texttt{cnd\_max\_employees}}\texttt{(d));}$$

where `d` is the department for which the rule is activated. If updates are treated separately (as presented in section 6.17) then there will be three partial differentials for $\Delta\texttt{make\_bag}_{\texttt{cnd\_max\_employees}}$ and the materialized aggregate function $\texttt{count}_{\texttt{cnd\_max\_employees}}$ will take four arguments for the bags of added, removed, updated tuples, and the department, respectively.

Partial differencing of the `sum` aggregate can be done in a similar manner as for `count`.

In [100] some work can be found on incremental maintenance of views with aggregates.

6.19 Rule Termination Analysis

Another problem with rule execution is that the actions of one rule can affect the conditions of other rules. This gives a potential risk for non-termination and unpredictable execution times of rules. Some work on rule analysis has been done such as in [8] where graphs are constructed on how rules affect each other and where cycles are detected.

Another simple way to avoid non-termination is to limit the number of rule execution cycles in the rule check phase. This is the approach currently chosen in AMOS. An error is signalled if the number of iterations exceeds a constant which can be set by the user. This, however, does not directly help the user with what to do if this occurs. A back-trace facility would here be helpful to let the user see which rules caused the problem. To detect possible sources of non-termination during the design phase it is desirable to have some analysis tools that can detect potential rule execution cycles.

6.20 Real-time Aspects of Rule Execution

An ADBMS involved in monitoring and control of real-time applications such as CIM and telecommunication network management needs to have some support for real-time behaviour. The rule priority specification at rule activation in AMOS should not be considered for achieving different performance of rule execution, but is rather for defining how simultaneously triggered rules should behave semantically, e.g. to avoid incorrect or non-terminating execution.

Real-time issues such as predictability and meeting deadlines are not dis-

cussed in this thesis. The work in this thesis is focused on active rule execution performance in general. Real-time issues considered in *real-time DBMSs* [101] can be applied to ADBMSs as well. Scheduling transactions triggered by active rules is not much different from scheduling any transactions where real-time limitations have to be considered. There are, however, some problems if the scheduling has to be done in advance to ensure that all deadlines will be met before execution has even begun. Such applications are usually hard real-time applications, e.g. fine-grained control loops in real-time process control systems, and it is questionable whether full-fledged ADBMSs (supporting complex rules in a declarative query language) should be used in such applications at all.

Predictability of access of data can also be applied to data accessed by rules. Ideas presented in [95] where new cost information based on quality of information and cost in access time guides the optimizer to choose different access methods, can be directly applied to query optimization of rule conditions.

There is also some research specific to active database systems where active rules are directly specified with real-time constraints, such as in REACH [18] and in [11].

7 The Propagation Network

7.1 Implementing Active Rules

Active rules can be implemented outside a DBMS as a wrapper application, but this has serious implications on functionality and performance as was explained in chapter 1. This chapter discusses detailed issues of implementing active rules in an ADBMS. The issues discussed include: capturing and storing events, building the propagation network, activating/deactivating rules, and the algorithm for propagating events and changes in the propagation network. Compilation and execution of rules using partial differencing were discussed in chapter 6.

7.2 Capturing and Storing Events

Most events in an ADBMS are related to operations that modify the database, such as changes to the contents of stored functions (tables) or changes to the database schema[1]. These operations are usually logged, i.e. stored in a transaction log, until the transaction is committed or aborted. If these operations affect an activated rule, i.e. a rule that references the operation directly in the event part or indirectly in the condition part, then the operation along with any operation data must be stored as an event. In the ADBMS manifesto [35] it is also stated that the transaction time of the operation must be recorded with the event as well.

A major difference between transaction logs and event logs (or event histories) is that a transaction log of disk-based DBMSs is usually a physical log that records changes to pages while an event log is a logical log that records events that represent logical changes to the database (e.g. tuples added to a table). Storing events can be performed by recording the operations along with the transaction time of their occurrence. In a main-memory DBMS such as AMOS a *logical transaction log* is maintained since recording physical changes to main-memory is difficult and usually inefficient. In some systems, a separate logical transaction log is maintained, such as the transition log in Starburst [84], which contains the modifications done in a transaction. These logical transaction logs can also serve as an event histories. Such logical

1. Most relational DBMSs store the schema in meta-tables so schema changes can sometimes be regarded as changes to tables as well.

transaction logs are usually special data structures that can be updated efficiently since new events will always be added last.

The logical transaction log will store all the events in one data structure and will make the access of particular events, e.g. the update events of a particular table, less efficient. Indexes can be added for accessing particular events, but this will make the insertion of events less efficient. Storing all events in one log makes it difficult to support different event consumption modes where all events are not consumed at the same time.

A second better alternative is to provide a *differentiated event log* by using partial event logs for each event type and table. This alternative was chosen for implementing events histories in AMOS (a similar technique is used in Monet [79]). The advantages compared to one event log are:

- Fast access since each partial event log is usually small.
- Supports efficient rule condition execution by using partial differencing.
- Events can be defined on views as well since partial differencing can be used to generate view events based on the events that the view depends on.
- Supports rule contexts that need events that exist longer than just during the transaction when they were raised. Each context will have its own partial event logs for storing the events of the rules activated into that particular context.
- Supports event consumption by partitioning the event logs into separate logs for each rule context that is currently active. When all the rules activated in a context have been checked, i.e. the events are considered to be consumed, then the event logs of that context are cleared (this is part of the propagation algorithm).
- Supports the concept of foreign events (see section 9.4.1) which will have their own partial event logs and which reflect the updates of a foreign data source that are not logged on the transaction log.

The Δ-functions Δ_{+}(add event), Δ_{-}(remove event), and Δ_{-+}(update event) were presented in chapter 6. These are temporal functions that directly access the differentiated event log. See chapter 8 for a discussion on temporal functions. Partial event logs are created for foreign events and can be used in partial differencing in the same manner as local events. The event logs (or event histories) are implemented as time series as presented in chapter 8. This allows for using all special indexing techniques defined for the time series. Foreign events can be defined as time windows where a maximum size is defined and outdated events are automatically discarded. Event logs are only created for events that are referenced in rules and events are only stored for activated rules in activated rule contexts.

7.3 The Propagation Network

The propagation network contains all the information needed to propagate changes affecting activated rules.

Let us look at the rule no_high again:

```
create rule no_high(department d) as
    when for each employee e
    where dept(e) = d and
        employee.netincome->number(e) > netincome(mgr(e))
    do set employee.grossincome->number(e) =
                                        grossincome(mgr(e));
```

The condition function for no_high is defined as:

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

The partial differencing technique is then used to define the partial differencing functions:
Δcnd_no_high/Δ_{-+}employee.netincome->number,

Δcnd_no_high/Δ_{-+}manager.netincome->number,

Δemployee.netincome->number/
Δ_{-+}employee.grossincome->number,

Δemployee.netincome->number/
Δ_{-+}employee.grossincome->number,

Δmanager.netincome->number/
Δ_{-+}manager.grossincome->number,

Δcnd_no_high/Δ_{-+}dept,

Δemployee.grossincome->number/Δ_{-+}income, and

Δmanager.grossincome->number/Δ_{-+}income.

When this rule is activated the above partial differentials will be inserted into the propagation network. The dependency graph representing the propagation network of cnd_no_high can be seen in fig. 6.17.

Since the propagation is performed in a breadth-first, bottom-up manner the network levels can be modelled as a sequential lists, starting with the lowest level and moving upwards. Each level consists of:

- A *change flag*, chg_flg, that marks a level as changed.
- A *list of network nodes*.

In fig. 7.1 the propagation network for the rule no_high can be seen showing

the different levels.

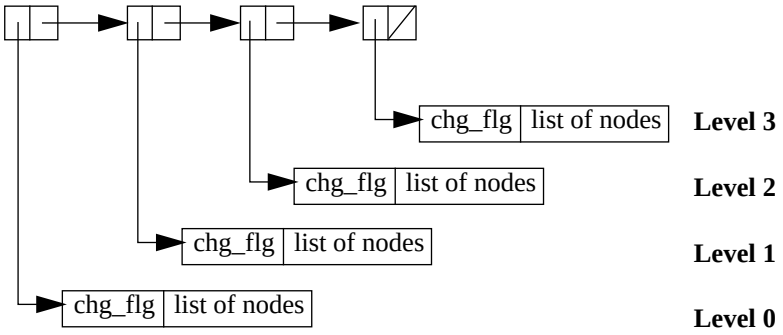


Figure 7.1: The propagation network for `no_high`

Each differential (with corresponding Δ-set) that can affect activated rules is associated with one (and only one) node (see fig. 7.2) consisting of:

- A *change flag*, chg_flg, marking the node as changed.
- An *event count*, event_cnt, that states how many event nodes (nodes with an event function) are dependent on this node (including itself).
- The *differential with corresponding Δ-set* or an *event function* with event_res associated with each rule activation for storing results from the execution of the event function.
- A *list of affects nodes*, a-list, that are affected by changes to this node.
- A *list of depends on nodes*, d-list, together with the partial differentials affected by the nodes below.
- A *pointer to the level* the node belongs to (not shown in fig. 7.2).
- A *reference counter* ref_cnt used by the propagation algorithm to determine when a Δ-set has been propagated to all nodes in the a-list and thus can be cleared (not shown in fig. 7.2).
- A *list of pointers to rule activations*, acts, which will also be in the conflict set if rules are triggered (not shown in fig. 7.2).

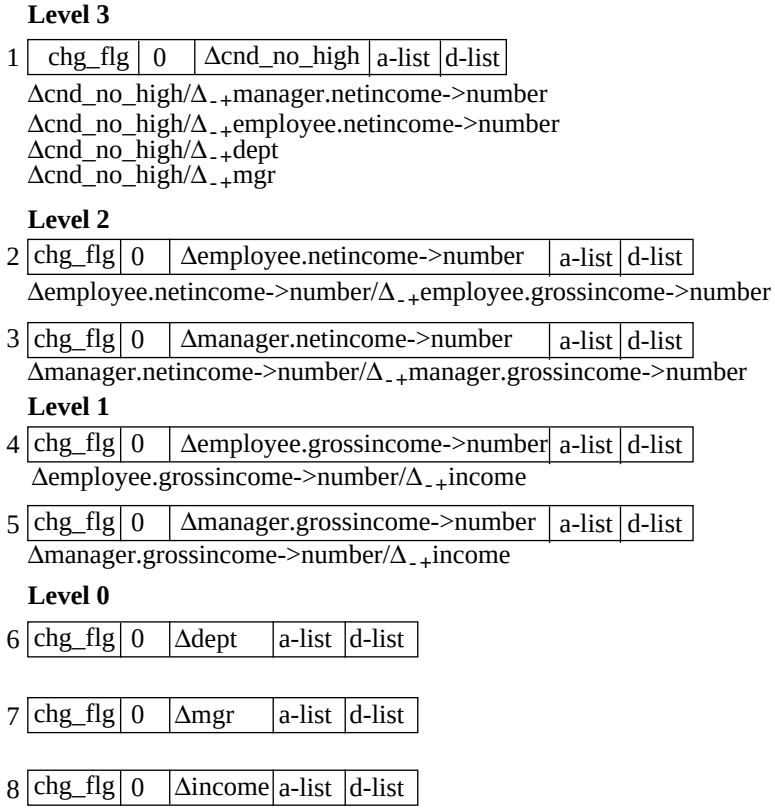


Figure 7.2: The nodes in the propagation network for `no_high`

The number of levels needed in a network depends on how relations are expanded. In the `no_high` rule there is overloading on `netincome` and `grossincome` that is resolved at compile-time (early binding). In the actual implementation the query plan will be expanded to directly use the `income` function, but this is disregarded in this discussion since it will make the propagation network flat and too trivial for explaining how it is constructed. Here the overloaded functions will cause extra nodes to be inserted in to the propagation network. In fig. 7.2 the four levels of nodes with their respective partial differentials can be seen. How the nodes in the four levels are connected for `no_high` can be seen in fig. 7.3 and fig. 7.4. In fig. 7.3 the upward dependencies, i.e. the a-lists, can be seen. It specifies what nodes above are dependent on changes to a Δ-set in a certain node. Note that the a-list of the top node has a NULL value since it has no upward dependency.

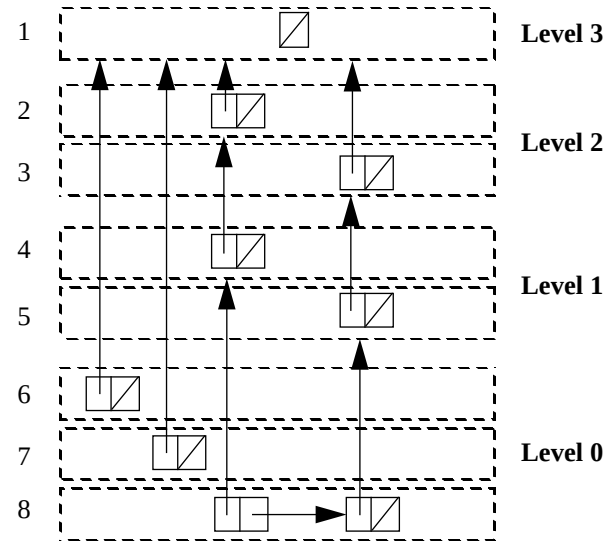


Figure 7.3: The a-list (affects) for upward dependencies for `no_high`

In fig. 7.4 the downward dependencies, i.e. the d-lists, can be seen. It specifies what changes in the Δ-sets of the nodes below a certain node is dependent on. Note that the d-lists for the nodes in level 0 have NULL values since they are not dependent on any other nodes (because they represent base events such as updates to stored functions). The a-lists and the d-lists are implemented as lists with direct pointers to the nodes. This is straight-forward to implement in a main-memory DBMS like AMOS. In a disk-based DBMS the propagation network has to be implemented with similar techniques as any other data structure in the DBMS such as B-trees. Propagation networks can become fairly large when there are many rules activated and with rules with fairly complicated conditions activated simultaneously. When large propagation networks are stored on disk, issues such as clustering have to be considered.

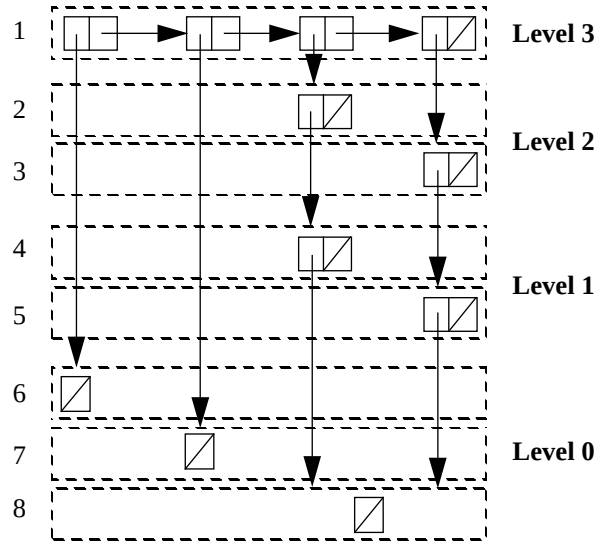


Figure 7.4: The d-list (depends on) for downward dependencies for `no_high`

For the late bound version of the rule `no_high` the propagation network will be expanded with an extra level for the extra node Δ`netincome`$_{\texttt{DTR}}$ (as discussed in section 6.13).

```
create rule no_high(department d) as
    when for each employee e
    where dept(e) = d and
        netincome(e) > netincome(mgr(e))
    do set employee.grossincome->number(e) =
                                    grossincome(mgr(e));
```

The condition function for the late bound `no_high` is defined as:

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

The dependency graph representing the propagation network can be seen in fig. 6.18. The propagation network nodes, a-lists, and d-lists can be seen in fig. 7.5, fig. 7.6, and fig. 7.7. Note that the node for Δ`netincome`$_{\texttt{DTR}}$ has no partial differential, instead it directly accesses the changes in the Δ-sets of its resolvent

differentials Δ`employee.netincome->number/`
Δ-+`employee.grossincome->number` and
Δ`manager.netincome->number/`
Δ-+`manager.grossincome->number`.

Which changes are accessed by partial differential Δc`nd_no_high/`Δ-+$\texttt{netincome}_{\texttt{DTR}}$ is determined at run-time (late binding). Note also that there is still a partial differential Δc`nd_no_high/`Δ-+`manager.netincome->number` since the `manager.netincome->number` is also referenced using early binding.

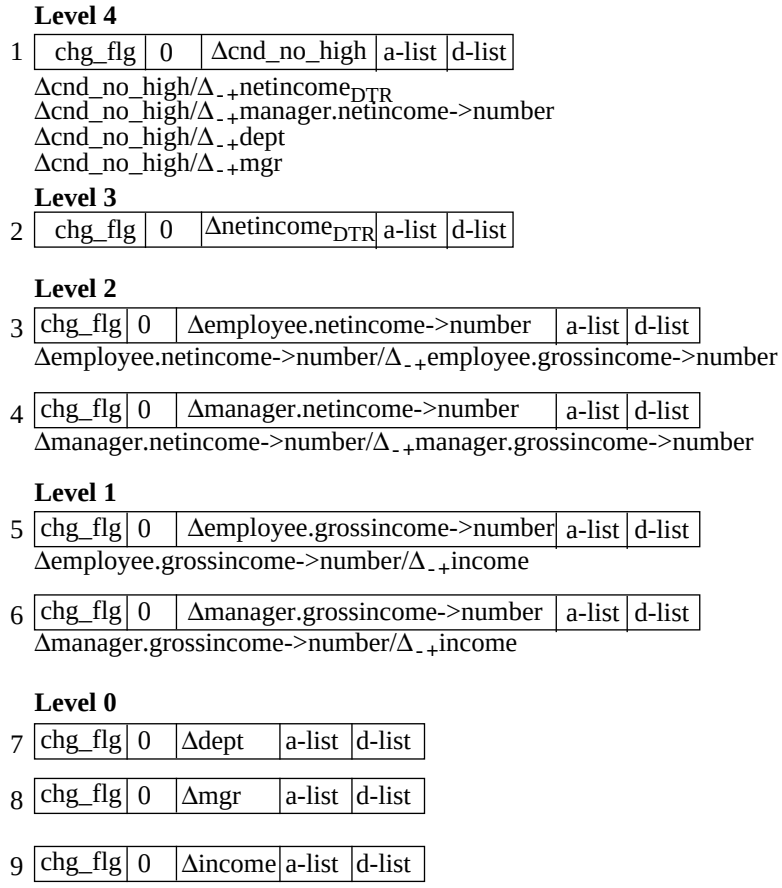


Figure 7.5: The nodes in the propagation network for late bound `no_high`

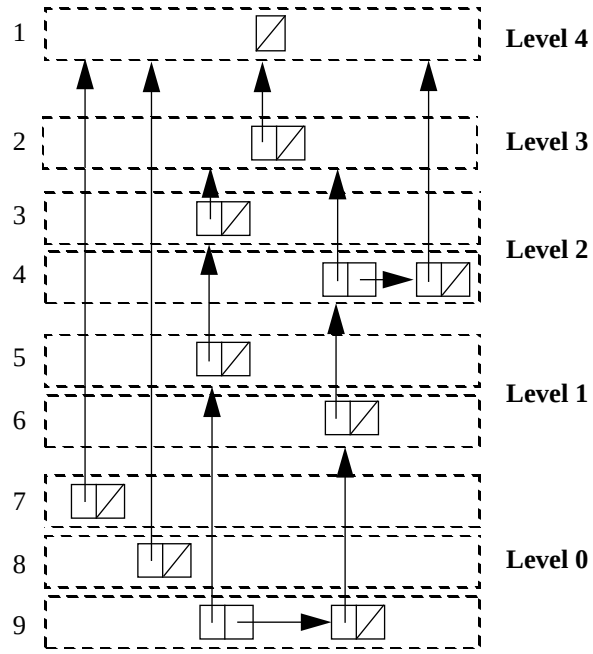


Figure 7.6: The a-list (affects) for upward dependencies for late bound `no_high`

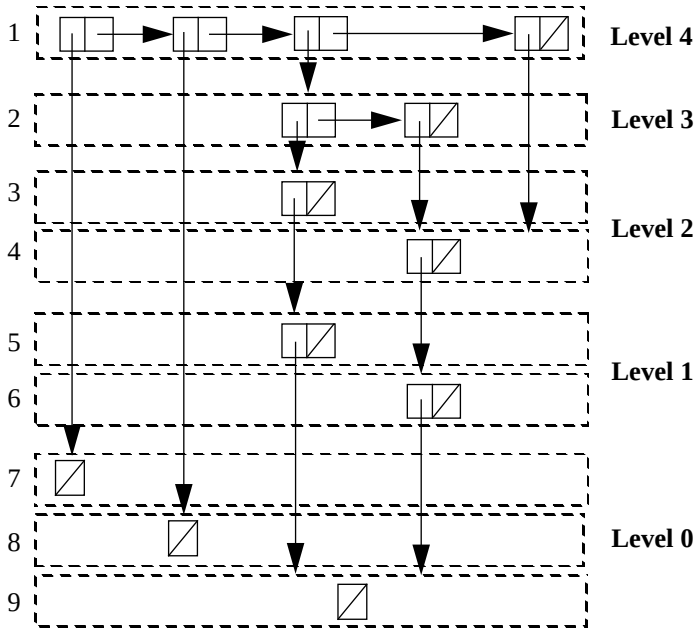


Figure 7.7: The d-list for downward dependencies for late bound `no_high`

The propagation network is not only used for propagating changes to rule conditions, but also for propagating the events for ECA and EA-rules as well. Let us look at the check_new rule again:

```
create rule check_new(department d) as
    for each employee e, manager m
    on updated(dept(e)) and
       updated(employee.netincome->number(e))
    when dept(e) = d and
         m = mgr(e) and
         employee.netincome->number(e) > netincome(m)
    do rollback;
```

The event function for check_new is defined as:

```
create function evt_check_new(department d_new)
                        -> <department d_new, employee e>
  as select d_new, e
  for each bag of department d_old,
           bag of number n_old, number n_new
   where <d_old, d_new> = Δ-+dept(e) and
         <n_old, n_new> = Δ-+employee.netincome->number(e);
```

The condition function of check_new is defined as:

```
create function cnd_check_new(department d, employee e)
                                    -> boolean as
    select true for each manager m
    where dept(e) = d and
         m = mgr(e) and
         employee.netincome->number(e) > netincome(m);
```

The dependency graph used for building the propagation network can be seen in fig. 6.22. The propagation network (fig. 7.8, fig. 7.9, and fig. 7.10) is now augmented with a special node (node 1) where the Δ-set is substituted with an event function. Note that the event count is now set for the nodes that propagates physical events to the event function.

Note that the partial differencing can still be used for evaluating the condition. In the check_new rule this is actually not needed since reasonable efficiency will be achieved by just passing the result from the event function to the condition function. The employee.netincome->number(e) in the condition function can now be executed efficiently since the employee argument e is now bound. This could be decided by the rule compiler which then would only generate the nodes for partial differencing of the condition function if they are

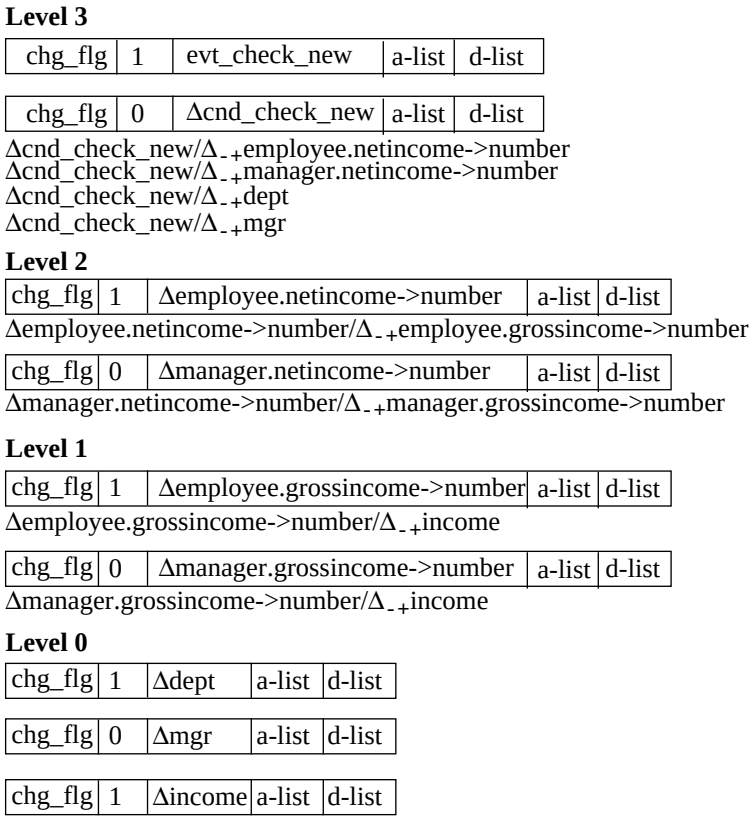


Figure 7.8: The nodes in the propagation network for `check_new`

to be used. In (fig. 7.8, fig. 7.9, and fig. 7.10) the propagation network for `check_new` which propagates both the events to the event function `evt_check_new` and changes to the condition function `Δcnd_check_new` can be seen.

When the partial differentials of `Δcnd_check_new` are executed, the department `d` and the employee `e` will be bound since they were passed from the event function. In this example incremental evaluation of the rule condition will not give much improved efficiency. In general, however, ECA-rules can contain complex conditions where the data passed from the event function cannot be used directly.

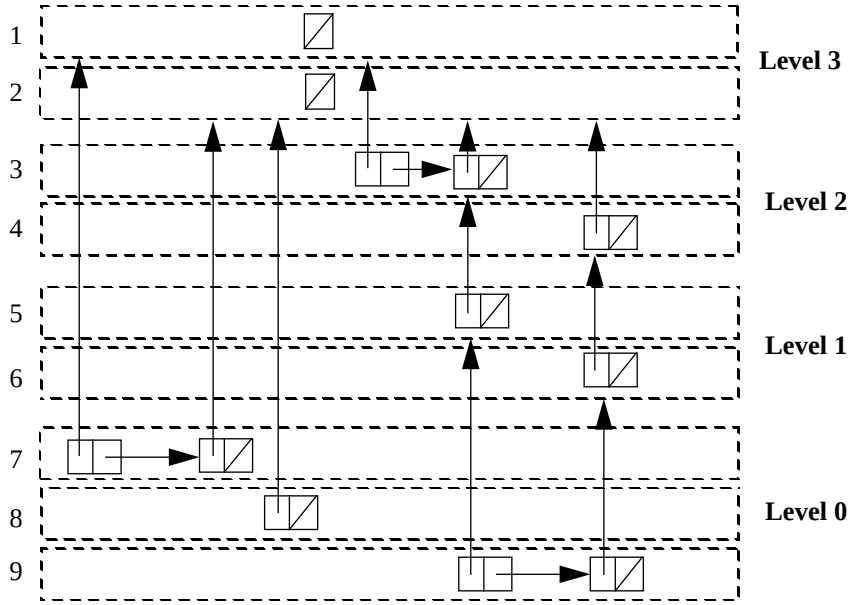


Figure 7.9: The a-list (affects) for upward dependencies for `check_new`

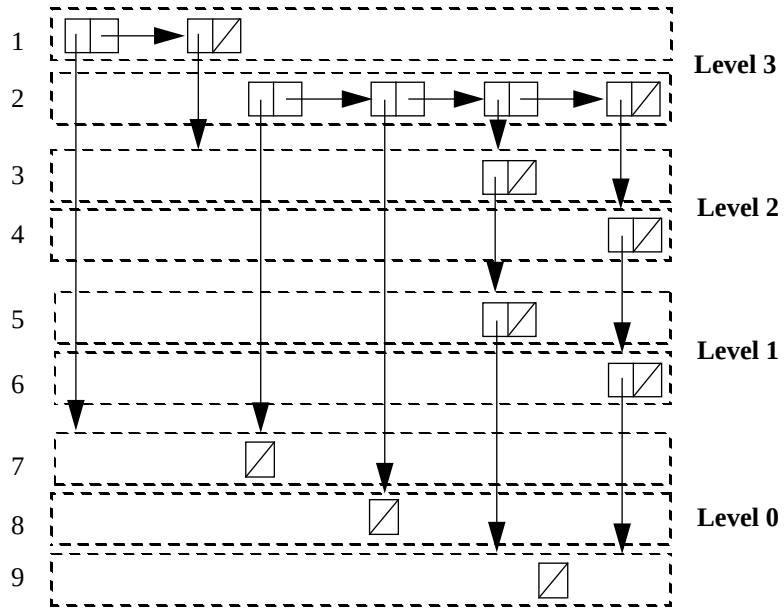


Figure 7.10: The d-list (depends on) for downward dependencies for `check_new`

If, for example, the rule `check_new` looks like:

```
create rule check_new(department d) as
    for each employee e, manager m
    on updated(dept(e)) and updated(income(e))
    when dept(e) = d and
         m = mgr(e) and
         mgr_makes_most(m) != true
    do rollback;
```

and where `mgr_makes_most` is defined as:

```
create function mgr_makes_most(manager m) -> boolean as
    select employee.netincome->number(e) < netincome(m)
    for each employee e where m = mgr(e);
```

In this rule condition the employee e from the event part is of no use for evaluating the `mgr_makes_most` function. With incremental evaluation of `mgr_makes_most` and using change propagation with partial differencing, the efficiency will be comparable to the first version of the `check_new` rule. The rule compiler will have to use the cost information calculated by the query optimizer when optimizing the condition function to determine whether partial differencing should be used or not.

Another example is the rule:

```
create rule check_total(department d) as
    for each employee e
    on updated(income(e))
    when sum(select income(e) for each employee e
                where dept(e) = d) >
                                    salary_account(d)
    do rollback;
```

This rule states that a given department must balance its salary budget. In this rule condition the sub-query in the call to `sum` contains a quantification of all employees. The `employee e` in the event part is not the same as the one specified in the sub-query in the condition (the new definition of `employee e` *shadows* the previous declaration). In this case the ECA-rule would be no more efficient than a CA-rule since the data from the event part cannot be passed directly to the condition. If the previous value of the sum is materialized (cached) then the change to the income of the employee specified in the event part can be used to manually calculate the new sum incrementally. A CA-rule that uses partial differencing and automatically does view maintenance of the cached `sum` (in a similar manner as for the incremental calculation of `count` in section 6.18) would here be the best solution.

7.4 Accessing Event Functions in Conditions and Actions

In section 5.6.3 a rule was presented that accessed an event function in the action part. In general, all event functions that are used for calculating the event part of an ECA-rule can be accessed in the condition and action parts. These may be references to added or removed tuples and to previous values of updates. The explicit reference can also fetch the transaction time of the event (see section 8.8).

When ECA-rules (and EA-rules) are compiled, the conditions and actions are analyzed for direct access to event functions. If the event functions are referenced in the condition or action parts, the event function is defined to return the contents of the referenced event functions to the condition and action functions. The conditions of ECA-rules do not directly access any explicitly referenced event functions themselves since they are accessed by the event function. The condition will implicitly access the event functions through partial differentials, but these physical events are transformed into logical events. The actions are not allowed to directly access the event functions themselves since the events are consumed before the action is executed. If the condition or action of an ECA-rule tries to access an event function that is not referenced in the event part, then the rule compiler will generate an error.

CA-rules are not allowed to explicitly access event functions in conditions and actions at all since CA-rules only monitor logical conditions. The partial differentials of the condition will implicitly access the event functions, but the physical events will be transformed into logical events.

7.5 Creation and Deletion of Rules

When a rule is created, an event function (if ECA or EA-rule), a condition function (if ECA or CA-rule), and an action procedure are created. When rules are created all partial differentials needed to monitor the rule condition are also generated (if they do not already exist) and optimized. When a rule is deleted, the event function, the condition function, the action procedure, and the partial differentials are also deleted (partial differentials shared with other rules will only be deleted when the last rule that uses them is deleted). Note also that rules cannot be deleted until they have been completely deactivated (with all actual arguments and from all rule contexts).

7.6 The Algorithms for Activating and Deactivating Rules

When an active rule is activated, it is inserted into the propagation network. If the rule is parameterized, then the rule is only inserted at the first activation. When a rule is deactivated, it is removed from the propagation network. If the rule is parameterized, it is only removed when the last activation of the rule, i.e. the last actual parameter pattern, is deactivated.

When a rule is to be inserted into the propagation network, the event func-

tion (if any) and the condition function have to be inserted by extending the propagation network with new nodes. When rules are activated/deactivated the network is expanded/contracted with/without the nodes needed to propagate changes to the currently active rule conditions. Two functions are needed for expanding/contracting the propagation network: one for inserting differentials or event functions (Insert) and one for removing differentials or event functions (Remove).

A differential Δf_c of a rule condition function f_c is inserted by Insert(network, Δf_c, false) and is removed by Remove(network, Δf_c, false). A rule event function f_e is inserted by Insert(network, f_e, true) and is removed by Remove(network, f_e, true).

The algorithm for inserting differentials and event functions into the network is a depth-first, top-down algorithm as follows:

```
Insert(network, ΔP, event_flg):
    if ΔP is not already inserted into the network then
        create node_of(ΔP);
        if event_flg
                then set event_cnt(node_of(ΔP)) = 1
                else set event_cnt(node_of(ΔP)) = 0;
        if D_P is empty, where D_P is the set of functions that P depends on,
        then   /* P is a stored function */
                Insert_in_level(network, node_of(ΔP), 0, event_flg);
        else /* P is a derived function */
                for each ΔQ where Q∈ D_P do
                        Insert(network, ΔQ, event_flg); /* recursive call */
                        insert (node_of(ΔQ) . ΔP/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(network, node_of(ΔP),
                    max(for each ΔQ where Q∈ D_P: level_of(node_of(ΔQ))) + 1,
                    event_flg)
    else /* ΔP is already in the network */
        if event_flg then
            set event_cnt(node_of(ΔP)) = event_cnt(node_of(ΔP)) + 1;
            /* increase the event_cnt of the nodes below */
            for each ΔQ where Q∈ D_P do Insert(network, ΔQ, event_flg);

Insert_in_level(network, node, level, event_flg):
    if level does not exist in network then create level;
    if event_flg
        then insert node first into the level of the network
        else insert node last into the level of the network;
    set level_of(node) = level;
```

The algorithm for removing differentials and event functions from the network is also a depth-first, top-down algorithm that looks as follows:

```
Remove(network, ΔP, event_flg):
    if ΔP is present the network then
        if the affects list node_of(ΔP).a-list is empty then
            for each ΔQ where Q∈ D_P
                remove (node_of(ΔQ) . ΔP/ΔQ) from the
                depends-on list node_of(ΔP).d-list;
                remove node_of(ΔP) from the affects list node_of(ΔQ).a-list;
                Remove(network, ΔQ, event_flg); /* recursive call */
            Remove_from_level(network, node_of(ΔP), level_of(node_of(ΔP)));
            delete node_of(ΔP)
        else /* the affects list node_of(ΔP).a-list is not empty */
            if event_flg then
                set event_cnt(node_of(ΔP)) = event_cnt(node_of(ΔP)) - 1;
                /* decrease the event_cnt of the nodes below */
                for each ΔQ where Q∈ D_P do Remove(network, ΔQ, event_flg);

Remove_from_level(network, node, level):
    remove node from level of network;
    if no nodes remain in the level then delete the level;
```

All operations to the network are transactional, i.e. the changes are logged so that they can be undone during a transaction rollback. This means that rules created/deleted during a transaction will be deleted/recreated if the transaction is rolled back and rules activated/deactivated during a transaction will be deactivated/reactivated if the transaction is rolled back.

Note that event functions are always placed first in each level. This is needed in the propagation algorithm which executes breadth-first (left to right in each level), bottom-up to always execute the event part of a rule before the condition (if the propagation network is used for propagating events to the event part as well as changes to the condition). See section 7.7 for a detailed description of the propagation algorithm.

When a CA-rule is activated a complete evaluation of the rule condition for the specific activation pattern is performed. The result is saved in the action-set of the rule. This is done in order for the rule to catch up with all the changes affecting it and that have occurred prior to the rule activation.

7.7 The Event and Change Propagation Algorithm

During ongoing transactions all changes to the logical transaction log are screened for changes that might affect activated rule conditions. If a change is made to a stored relation that has a corresponding node in level 0 in the propagation network, i.e. if a relevant update event is detected, then the change is

added to the corresponding Δ-set (using $\cup_\Delta$ if the event_cnt of the corresponding node is 0 and using $\cup$ otherwise). The node of the changed Δ-set is marked as changed (chg_flg of both the network level and the node) and the reference count (ref_cnt) is set to the length of the a-list of the node.

In the check phase the propagation algorithm propagates all the non-empty Δ-sets in a breadth-first manner, as illustrated in fig. 6.9. 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, bottom-up propagation ensures that all the changes have been calculated when we reach that node.

In the check phase one round of propagation is first done using the changes accumulated in Δ-sets throughout the transaction (fig. 7.11). If any rules were triggered, i.e. were inserted into the conflict set in the propagation, then one rule activation is chosen, using some conflict resolution method. The action part of the chosen rule activation is then executed using the tuples generated in the condition of the rule. The action part is executed for each positive change since the last check phase, we call this the *action set*, which is calculated from the Δ-set of the condition (passing the positive data from Δ_+ and Δ_{-+} to the action function). To determine if an already triggered rule (i.e. it is in the conflict set) is no longer triggered, the action set is saved and is modified continuously using the Δ-set of the condition (removing any negative data found in Δ_- and Δ_{-+}) to determine if it is still triggered. This is only done for CA-rules that monitor logical changes (and negative) changes to their conditions, i.e. conditions with negative partial differentials that are executed during the change propagation.

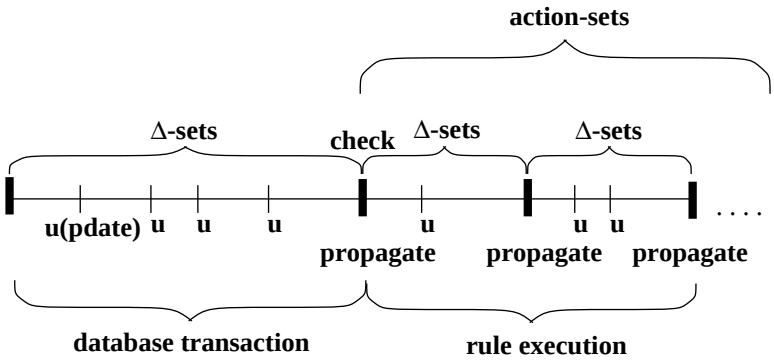


Figure 7.11: The propagation phases in rule execution

If a rule is triggered, a *rule activation* is inserted into the conflict-set. A rule activation consists of:

- The *rule* that was activated.
- The specific rule *activation arguments*.

- The *result of the associated event function*, event_res (used for temporary storage in the propagation algorithm).
- The *action set* which contains the tuples on which the action is to be applied.

The rule execution phase continues to propagate, trigger, and execute rules until no more events (empty Δ-sets) are detected. The algorithm presented here is not dependent on any specific conflict resolution method. In the present implementation of rules in AMOS a simple priority scheme is used. To support this, each rule activation has a priority and the conflict set is divided into several priority levels. When a rule activation is triggered it is inserted into the corresponding priority level of the conflict-set. If the condition for which a rule was triggered changes to false, i.e. the action-set of a rule activation becomes empty, then the rule activation will be removed from the conflict-set without executing the action.

The rule execution works in four stages (see fig. 7.12):

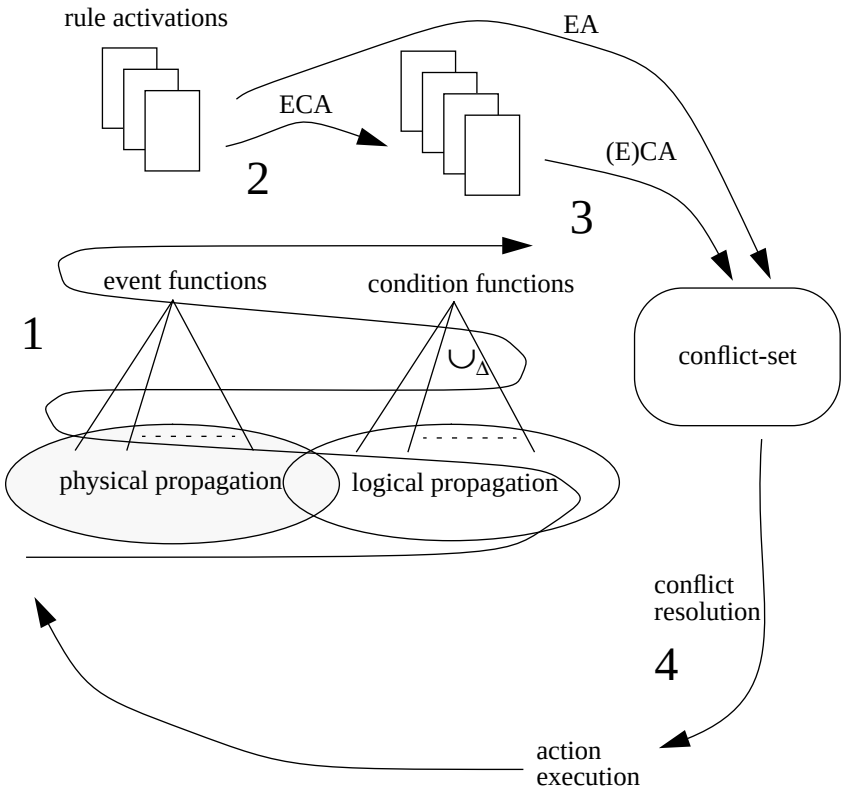


Figure 7.12: The four stages of rule execution

1. The events and changes are propagated in the propagation network.
2. The event parts are evaluated for each rule activation and the results are saved.
3. The event data (if ECA-rule) is used for evaluation the of the rule conditions. Data from rule conditions which are non-empty is stored in action-sets for the corresponding rule activations and which are stored in the conflict-set. EA-rules store their event data directly with the corresponding rule activations in the conflict-set.
4. The rule activation with the highest priority chosen from the conflict-set and its action is executed.

The four stages are repeated until the conflict-set is empty. The propagation algorithm is defined using three functions:

- propagate(network) that does the actual breadth-first, bottom-up propagation.
- evaluate_ruleEC(node) that evaluates event and condition functions of a top node in the propagation network.
- trigger_rule(activation, Δ-set) that takes the changes to the condition function and triggers or untriggers the rule activation.

The propagation algorithm looks as follows:

```
propagate(network):
  for each level in the network do /* starting with level 0 */
    if level.chg_flg then /* some node in the level has changed */
      for each node in level.nodes do /*moving left to right */
        if node.chg_flg then /* we have found a changed node */
          if node.a-list is empty then
            /* node is a top node */
            evaluate_ruleEC(node)
          else /* not a top node */
            for each below-node in node.d-list do
              if below-node.Δ-set is non-empty then
                execute each partial differential and
                accumulate the result into the node.Δ-set
                (using ∪Δ if node.event_flg = 0 and ∪ otherwise);
                decrease_count(below-node);
            if node.Δ-set is not empty then
              for each above-node in node.a-list do
                set above-node.chg_flg = true;
                set (above-node.level).chg_flg = true;
              set node.ref_cnt = length(node.a-list);
          set node.chg_flg = false;
    level.chg_flag = false;
```

```
evaluate_ruleEC(node):
    for each rule activation in node.acts do
        if node is an event node then
            /* ECA- or EA-rule */
            execute the node event function using the activation arguments
            and Δ-sets in the nodes below and save the result in
            activation.event_res;
            if EA-rule and activation.event_res is non-empty then
                /* pass the activation.event_res directly to the action */
                trigger_rule(activation, <activation.event_res,{},{}>);
                clear activation.event_res;
        else
            /* a condition node */
            if ECA-rule then
                for each below-node in node.d-list do
                    if below-node.Δ-set is non-empty then
                        execute each partial differential
                        using activation.event_res as arguments and
                        accumulate the result into node.Δ-set  (using ∪Δ);
                clear activation.event_res;
            if CA-rule then
                for each below-node in node.d-list do
                    if below-node.Δ-set is non-empty then
                        execute each partial differential
                        using the activation arguments and
                        accumulate the result into node.Δ-set  (using ∪Δ);
            if node.Δ-set is non-empty then
                trigger_rule(activation, node.Δ-set);[1]
                clear node.Δ-set;[2]
    for each below-node in node.d-list do
        if below-node.Δ-set is non-empty then
            decrease_count(below-node);

trigger_rule(activation, Δ-set):
    if activation.action-set is empty then
        /* the rule activation is not previously triggered */
        set activation.action-set = Δ-set;
        insert activation into the conflict-set; /* using the activation priority */
    else /* the rule activation is already triggered */
        set activation.action-set = activation.action-set ∪Δ Δ-set;
```

1. In case of strict rule semantics a check should be made here that none of the tuples in the Δ-set are in the materialized view representing the old value of the condition function.
2. In case of strict rule semantics the Δ-set should here be used for maintaining the materialized view that represents the old value of the condition function.

```
        if activation.action-set is empty then
            /* the rule is not triggered */
            remove activation from the conflict-set;

decrease_count(node)
    set node.ref_cnt = node.ref_cnt - 1;
    if node.ref_cnt = 0 then
        clear node.Δ-set; /* deleting consumed events */
```

Note that the algorithm presented here does not handle recursion, but can be extended to handle this. Work on incremental evaluation of recursive expressions can be found in [65]. AMOSQL provides a *transitive closure*[1] operator that can handle most of the queries where recursive evaluation is needed. This operator is easier (or less difficult) to evaluate incrementally than general recursion since it involves looping over only one node in the network.

7.8 The Check Phase and Propagating Rule Contexts

The check phase is started either automatically just before a transaction is committed or if the user calls the check function. The check function is called with an optional rule context with the *deferred rule context* as the default (called automatically when a transaction is committed).

Rule contexts are groups of activated rules that are to be executed in the same check phase (see Paper V). Rules are activated into rule contexts and rule contexts are activated causing all of the active rules in them to be monitored, i.e. events affecting active rules in the context are accumulated in the differentiated event logs (Δ-sets) of the context. Likewise, rule contexts can be deactivated causing the monitoring of the active rules in them to stop, i.e. no events are accumulated for the context. Rules can also be deactivated from contexts. If rules are activated without mentioning any context then they are by default activated into the deferred rule context.

The rule contexts are implemented as separate objects of the type context. Each context has its own propagation network attached to it. When rules are activated into a context they are inserted into the propagation network of that context. When a context is activated, it is just marked as active. All event logs that an active context is dependent on, i.e. the event functions in the leaves of the propagation network, are indexed by the context to save the events specifically with that context. When events are captured, they are saved in separate event logs, i.e. event functions indexed by each active context. If there are no activated contexts that are interested in the event, then it is not saved.

When a context is deactivated, it is just marked not active. The propagation network is not modified during context activation and deactivation which makes it much faster than activating and deactivating separate rules. When

1. Transitive closure performs repetitive application of a function,
   $tclose(f_{function}, o_{object}, n_{integer}) = f^n(o)$

rules are deactivated from a context, they are removed from the propagation network. The algorithms for rule activation/deactivation presented in section 7.6 have an extra parameter for the specific network of the context that the rules are being activated into (or deactivated from).

When a context is to be checked, the changes are only propagated if the context is active. After the changes have been propagated, the event functions (i.e. event histories) of that context are cleared. Since each context has its own event functions, there is no problem of one context consuming the events needed by another context.

The check function looks as follows:

```
check(context):
    if context is active then
        propagate(network(context));
        while conflict-set is not empty
            choose the rule activation with the highest priority
            from the conflict-set;
            execute the action using the calculated action-set[1];
            clear the action-set of the executed rule activation;
            propagate(network(context));
```

In the implementation a check is made if the iteration in the check function exceeds a certain limit. If this happens the check phase is aborted with an error stating there is probably a non-termination problem with the current set of active rules.

7.9 Event Consumption

Event consumption usually considers when events are consumed by rules during event monitoring. AMOS has a very simple event consumption model and does not support user-defined consumption models like Chimera [133]. In AMOS events are consumed when they have been processed for all rules that are monitoring them. To be more precise, when the Δ-sets have been propagated to all nodes above in the propagation network that depends on them, i.e. when any partial differentials or event functions that access them have been executed, the Δ-sets are cleared. Event and condition functions generate tuples that are stored in action-sets associated with each active rule. The action-sets are maintained, i.e. expanded by added tuples and contracted by removed tuples, for triggered rules and are cleared after the action is executed.

All rules that have been activated into a rule context are processed at the same time, i.e. in the same propagation phase. Different contexts have their own copies of Δ-sets so they cannot consume each other's events. Rule activations are unique for each rule context and thus have their own action-sets. The same rule can be activated with the same arguments (or different ones) into

1. Using the positive parts of Δ_+ and Δ_{-+}.

several contexts, but these will be represented by different rule activations with separate action-sets.

Time series are defined to support moving time windows where old data are discarded when they are considered too old. This is important for event functions for foreign data sources (see chapter 9) where the number of events can accumulate quickly. Time series defined with a maximum size will automatically discard the oldest data when they have reached their maximum size.

Time series are discussed more in chapter 8.

8 Time Series and Event Histories

8.1 Time in Applications and ADBMSs

Many applications use the concept of time. In CIM data originating from sensors need to be timestamped for the system to determine the usability (recency) of the data. Time series of sensor data can be used to determine trends such as the direction of moving of objects and if some sensor value such as temperature, is increasing or decreasing. In some advanced applications the sensor data is seen as a function varying over time and on which specific transformation algorithms will be executed as, for example, in stock market trend analysis.

In telecommunications time is important as well. In network management functions such as accounting, load statistics, and fault management are all based on information being timestamped. For example, it is important to know at what times during the day there are communication bottlenecks in the network and if the congestion levels in alarms from some network element are increasing or decreasing. To support applications such as CIM and telecommunication network management DBMSs need to have some support for time.

In an ADBMS it is desirable have active rules that can access the time when events occurred. For example, the event specification of an ECA-rule can dictate that one event must have occurred before or after another event. Rule conditions can contain special functions which perform special operations such as interpolation or extrapolation of time series that represent changes to data over time.

8.2 Temporal Databases and Scientific Databases

In *temporal databases* [111] the DBMS has been extended to support storing and accessing data through a temporal extension of the query language. The time dimension can be the time of insertion, temporal validity of data, or user-defined time. A temporal DBMS usually has extensions to the basic relational operations to support temporal queries, i.e. queries that use the time dimension.

In *statistical* and *scientific databases* [109][136] the DBMS has been extended to support storing data from results of scientific experiments. The data can represent discrete samples of continuous functions or measurements and is usually stored in time series with special support for operations such as

statistical operations (averages, mean values) and interpolation of discrete values to reconstruct a continuous function.

8.3 Supporting Time in Databases

When considering support for time in databases it is important to define what *timelines* should be supported. A common classification of different timelines is:

- *Transaction time*. When some data was inserted (and usually when committed) into the database (e.g. the employee salary table was updated at 12:01, September 2, 1996).
- *Valid time*. When some data stored in the database is (was) valid (e.g. the new salary of a specific employee is valid from or will take effect at, 00:00, September 3, 1996 until changed again). The valid time is usually different from transaction time, but in the case of storing changes of physical entities when the changes happens, the transaction time and valid time can be considered the same.
- *User time*. Some temporal data that is not directly supported by the database, but is still considered as temporal by the user (e.g. the new salary will start at the date agreed during negotiations).
- *Event time*. In active databases the time when an event occurred is important. For update events this will equal transaction time for uncommitted updates. Rules referencing time in the event part of a rule will presuppose event time.

Storing time in a database can be done by storing *time stamps* representing instant time values (single chronon), *time intervals*, or *time series* for transaction or valid time. For storing data with timelines of both transaction and valid time, *bitemporal chronons* can be used.

8.4 Time Stamps

A basic requirement is to support storing time as a data type. Storing time as character strings or as integers might be inefficient and does not support strong typing for operations specific on the time type. Absolute time in UNIX systems is usually defined as two integers, *timevals*, that represent the elapsed number of seconds and μ-seconds elapsed since 00:00, January 1, 1970. More work on storing time stamps in databases can be found in [37]. In AMOS time stamps are instances of the type `timeval` and are referenced as `|year-month-day/hour:minute:second|`. Internally AMOS timevals are stored as UNIX timevals using GMT, but with automatic translation to local time.

8.5 Time Intervals

Time intervals are usually pairs of timestamps representing the beginning and

end of an interval. The intervals can also be defined as open- or closed-ended. In AMOS intervals are referenced as `|[start-timeval,stop-timeval)|` with '[' representing a closed end of the interval and ')' representing an open end.

8.6 Time Series

Time series are discussed more in the area of scientific databases than in temporal databases. Time series can be classified as *sparse* (irregular) or *dense* (regular), where sparse means that some interval within a time series does not contain any data, while dense often refers to data generated with a constant time period between the data in the time series. Sparse time series have to be stored with the time-stamp for each datum, while timestamps in dense time series can be calculated from the time period. Time series can be both sparse and dense in different time intervals.

Time series (sparse with explicit timestamps) have been implemented in AMOS as a foreign data source with a special data structure for efficient storage and access. Functions for indexing based on timestamps and time intervals have been defined. Special operations have been defined on timeseries such as time series difference (-) and time series union ($\cup$) to support efficient execution of the $\cup_{\Delta}$ operation (see section 6.17).

8.7 Temporal Functions

Time series in AMOS have been used for implementing timestamp-based temporal functions. The timestamps for temporal functions usually reflect the transaction time. Temporal functions can be accessed just like any other function, but have efficient access based on timestamps or time intervals.

Temporal functions in AMOS can be accessed through several overloaded versions and using the `@` (at) operation for timestamp access and the `within` operation for time interval access.

For example, if we have a temporal function `department_meetings` that stores the time and a description of department meetings, we can access the function in several ways:

```
select t, c for each timeval t, charstring c
where c = department_metings(:toys_department)@t;
<|1996-12-01/15:00:00|, "Sales meeting">
<|1996-12-03/12:15:00|, "Lunch meeting">
<|1996-12-07/19:00:00|, "Late meeting">
<|1996-12-11/15:00:00|, "Sales meeting">

select department_metings(:toys_department);
"Sales meeting"
"Lunch meeting"
```

```
"Late meeting"
"Sales meeting"

select department_metings(:toys_department)@
                                  |1996-12-01/15:00:00|;
"Sales meeting"

select department_metings(:toys_department) within
       |(|1996-12-01/15:00:00|, |1996-12-07/19:00:00|]|;
"Lunch meeting"
"Late meeting"
```

The implementation in AMOS is not a complete temporal extension of AMOSQL, but is enough for supporting implementation of event functions as temporal functions. Currently temporal functions in AMOS are only represented by event functions that are used for storing events monitored by active rules. These functions are transaction time relations, i.e. the timestamps for event functions reflect the time when the event occurred in a transaction.

8.8 Time Stamped Events

In the active database manifesto [35] events are specified as a pair (<event type>, <time>) stored in an event log (event history) where the time specifies the transaction time when the event occurred. In AMOS this is achieved by storing the events in event functions defined as temporal functions. Take an ECA-rule such as:

```
create rule check_new(department d) as
    for each employee e, manager m
    on updated(dept(e)) and updated(income(e))
    when dept(e) = d and
         m = mgr(e) and
         netincome(e) > netincome(m)
    do rollback;
```

Here the event functions for `dept` and `income` are stored as time series. The event function for the rule is really defined as:

```
create function evt_check_new(department d)
                        -> <department d_new, employee e>
    as select d, e
    for each bag of department d_old,
              bag of number n_old, number n_new,
              timeval t1, timeval t2
    where <d_old, d_new> = Δ-+dept(e)@t1 and
          <n_old, n_new> = Δ-+income(e)@t2;
```

The event functions can be referenced in the condition so the time of an event can be easily retrieved. Event functions can be accessed by using time intervals which is useful for relating events in time. A rule such as:

```
create rule check_new(department d) as
    for each employee e, manager m
    on updated(dept(e)) before updated(income(e))
    when dept(e) = d and
         m = mgr(e) and
         netincome(e) > netincome(m)
    do rollback;
```

will have an event function that looks like:

```
create function evt_check_new(department d)
                        -> <department d_new, employee e>
    as select d, e
    for each bag of department d_old,
              bag of number n_old, number n_new,
              timeval t1, timeval t2, timeval t3
    where <d_old, d_new> = Δ-+dept(e) within |[t1,t2]| and
          <n_old, n_new> = Δ-+income(e) within |(t2,t3]|;
```

This expression is more efficient than accessing the event functions with just a timestamp since time series can use the start of an interval as the starting index for scanning.

8.9 About Paper VII

Time series are often accessed using inverse queries, i.e. using the values as keys instead of the timestamps. To support efficient inverse queries over time series a new indexing technique was developed. This could be useful in active rules that reference a temporal data source (represented by a long time serie) in the event part and where the actual time(s) for a specific value is/are needed in the condition part and the event time(s) cannot be passed from the event part to the condition part. It can also be useful if the condition uses interpolation when accessing the time serie and needs the time of an interpolated value.

8.10 Temporal Event Specifications and Temporal Conditions

It is also desirable to define active rules that directly reference time in event specifications and rule conditions. For example, an event specification might define that a rule condition should be checked *after* a certain time period that the event(s) occurred or the condition of a rule should *hold* true over a certain period of time. Some work on *temporal triggers* and ECA-rules where time can

be explicitly referenced can be found in [24][64][98][134]. When foreign data sources are introduced into an ADBMS it is desirable to have support for specifying complex temporal rules that monitor changes of the foreign data (see section 9.4.1).

9 Foreign Data Sources

9.1 Introduction

DBMSs are more and more being integrated in systems that produce data from various sources that need to be integrated into the database. In many cases the data should not be stored physically in the database, but should instead be accessed externally when referenced. We call such sources *foreign data sources*. These foreign data sources should be presented in the database as if they are local data and the database should support access (including optimization of queries involving access to foreign data sources), monitoring, e.g. using triggers or ECA-rules, and possibly updates in a transparent manner.

In applications such as manufacturing process control, telecommunications network management, and financial information systems new data comes from foreign data sources such as manufacturing equipment, network elements, or the stock exchange. This data might originate from physical sensors (e.g. a thermometer sensing the temperature in a chemical process), external pieces of software (e.g. real-time control software with functions returning the state of a controlled process) or as in the case of the stock exchange, a transaction system with its own databases. If we want to monitor changes to this data in an

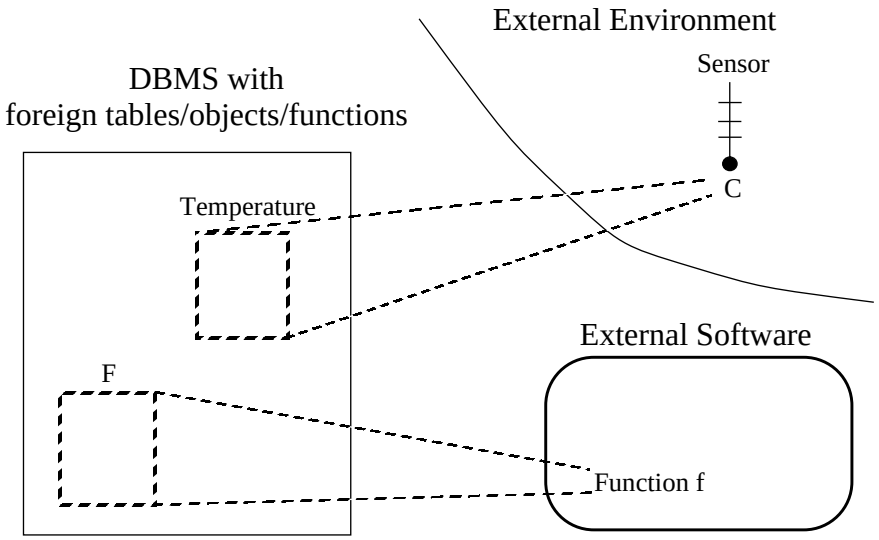


Figure 9.1: Foreign data sources in a DBMS

ADBMS, the traditional way is to store the data in the database and monitor it there when it changes. In applications such as the ones above, this is not a feasible solution since the amount of data is probably too large or is changing too frequently to be stored permanently, or it is already stored permanently somewhere else and it is unnecessary to store it twice. By allowing the database to refer to this data as foreign data sources (see fig. 9.1) and in the same manner as if it was stored in the database we can also handle it as local data, i.e. access it in queries, monitor changes to it, and perhaps update it.

Providing transparent access involves how to define foreign data sources in the database schema. The DBMSs must also define how to physically connect foreign data sources, e.g. defining communication protocols, polled or interrupt driven communication, synchronous or asynchronous communication, and any transformations needed to be done on the data. If the foreign data source is to be stored permanently or semi-permanently in the database, this has to be supported. If new data structures are defined in the database, these have to be supported with new data types with access methods and indexing techniques. Optimization of queries involving foreign data sources requires an extensible query optimizer. To support monitoring of foreign data sources mechanisms have to be defined as to how to inform the database that a foreign data source has changed, e.g. through signalling of events that can be referenced in triggers or ECA-rules. Updates of foreign data sources might not always be possible, e.g. if they represent information sensed in some physical environment. Sometimes updates can be made indirectly through user-defined update procedures.

To achieve transparency between foreign data and local data there are several issues that have to be addressed:

Defining foreign data sources in the database schema

If the goal is to provide real transparency , we must be able to define how foreign data is to be represented in the database and how to make it transparent to other stored data. This could be done as tables (in an RDBMS), as objects or object attributes (in an OODBMS), or as functions (in an ORDBMS [1 18]). Several pieces of information have to be provided to the DBMS for it to be able to access the foreign data source, e.g. data formats and means of communication.

Accessing foreign data

Transparent access to foreign tables, objects, or functions must be provided. If the foreign data is represented in some other format than can be supported or is required in the database, we must specify how the data should be translated. Standard protocols should be considered here, but must perhaps be extended. If we can support several dif ferent communication protocols, we might define which one the database should use for specific data sources. If the data is polled (using a pull model protocol, see section 9.3.4), we might want to specify polling frequencies. Data from foreign data sources must also be transferred to the database and perhaps be stored, permanently or semi-permanently (see

section 9.4.5). If the data is temporal, i.e. timestamped, and cached it might be useful to define how recent the data should be and automatically discard data which is too old. Special data structures might be needed for storing data such as time series with indexing for fast access. To access the foreign data the database must transparently communicate with the foreign data sources when they are referenced in database queries and in active rules. If the foreign data sources can interrupt the DBMS, the DBMS can be notified through OS interrupt signals and process updates in the background. The DBMS will be interrupted when foreign data has changed and the interrupt handler of the DBMS will start a background transaction that fetches the latest data and stores it in the database (see section 9.3.3). If the foreign data sources cannot interrupt the database, they have to be polled at regular intervals (see section 9.3.4). This processing can be done by database transactions running in the background without disturbing user or application transactions. A combination of both interrupts and polling is also possible. Caching the latest values in the database might also be possible here. If long communication delays are involved when communicating with foreign data sources, it is important not to block the system while waiting for replies (see section 9.3.5). Queries containing access to foreign data sources must be optimized based on the cost of accessing them. This must include communication delays, costs of translating data, and costs of searching data structures storing the foreign data.

Monitoring changes to foreign data

To support monitoring of foreign data sources, changes to the foreign data sources must be signalled as events that can be referenced in triggers or ECA-rules. (E)CA-rules with conditions that access foreign data sources makes it necessary for partial dif ferencing to support monitoring of changes to foreign data sources. Changes to foreign data sources can be defined in terms of Δ-sets to make them indistinguishable from changes to locally stored data. In the ECA-model, external events (foreign events) can be introduced that represent the change of some foreign data source (see section 9.4.1).

Updating foreign data sources

In order to provide full transparency between foreign and stored data we need to define what it means to update a foreign data source. If data is just stored externally the action could be to just change the externally stored value. If the value is represented by a sensor it might not be possible to change it directly , but perhaps indirectly . Changes to local data are logged and we allow changes to be undone by aborting transactions. This could be the case for externally stored data as well, but not for sensed data since it reflects a value in some external environment that we have no direct (but perhaps indirect) control over . Sensors can sometimes be indirectly updated by operating some actuator through a special update procedure (e.g. changing the temperature sensed by a thermometer by operating a heater).

9.2 Related Work

In Starburst [84] virtual tables can be defined that represent data stored outside the database.

In the Amazonia project [105] a framework was developed for providing transparent access of scientific data and various tools and services distributed over a network. Different access methods such as NFS, Anonymous FTP, and PFS (protocol for data access and data transfer) are supported. Support for specifying "filters" on the data as well as storing data locally in a cache to minimize communication is also provided.

The STRIP system [2] is a soft real-time main memory database with special facilities for importing, exporting, and handling derived data. The STRIP system exchanges data with other systems over *streams*. The actual format of the data is described in the stream schema record that is provided whenever a new connection is provided.

In the RAPID [134] DBMS external data produced by sensors is stored in *history tables* as time series with limited sizes. Changes to the history tables can be monitored with emphasis on fast response and using SQL3 [90] like triggers with temporal extensions.

In the TriggerMan [64] trigger processor external data stored in some database server (e.g. Informix, Oracle, or Sybase) can be defined and triggers can monitor changes to this external data. TriggerMan uses the replication server interfaces to monitor internal events representing changes to local tables in the database servers.

In OLE DB [14], new interfaces are defined with support for external data access (see section 9.5.3).

9.2.1 Commercial Replication Servers

In distributed databases it is possible to replicate tables on several nodes and to fragment the tables horizontally (row-wise) and vertically (column-wise). The replicated and fragmented tables can be accessed transparently as if they are local tables stored in the server that is accessing them (see fig. 9.2).

If a replicated table is updated, the DBMS makes sure that the master table and all the replicas are updated. This can be compared with the goal of introducing foreign data sources as tables where we want the tables to act as "replicas" of the state of the foreign data source (see fig. 9.1). In a distributed database the master copy of a table can be viewed as a foreign data source to all databases that keep replicas of the table. The actual techniques for implementing replication servers [113] can be used to implement foreign data sources.

A distributed DBMS can use the replication server interface to import foreign data into the database. This can be done by defining a foreign data source as a replication server that maps its data as 'replicated' tables and that supports the DBMS with access of the foreign data and informs about any changes to the data.

In many distributed DBMSs it is also possible to use the interface utilized

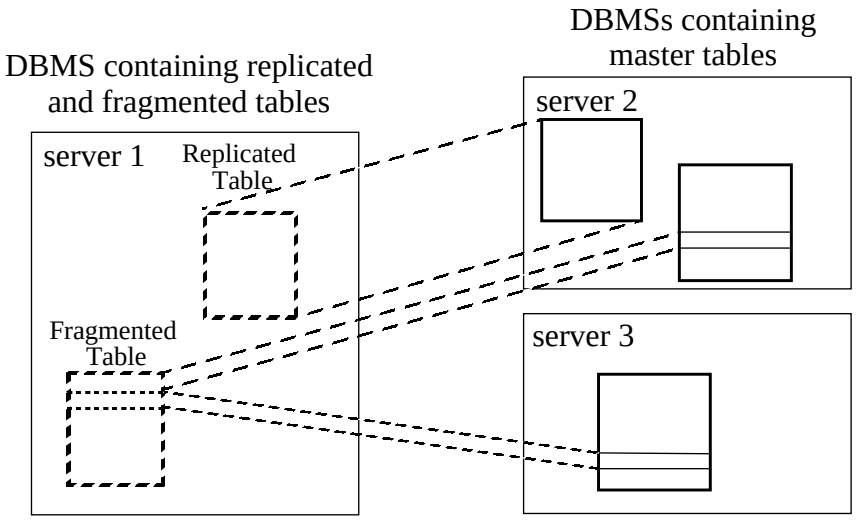


Figure 9.2: Replicated and fragmented tables

between replicated nodes to write a special application node as a replication server that can monitor changes to local tables in the DBMS. The interface passes information about updated tables (rows and attributes) making it possible to execute active rules outside the database server and monitor internal update events (even updates generated from the execution of stored procedures in the database server).

Below follows a short presentation of the techniques used by Sybase, IBM/DB2, and Oracle based on the comparisons in [113]. None of the systems discussed currently support extending the query optimizer to deal specially with foreign data sources. They do not currently support introducing any new special data structures to support storing foreign data either.

Sybase System 10’ s Replication Server

In Sybase System 10’s Replication Server all updates are committed to the master table before they are transmitted asynchronously to the sites containing the replicas. Remote sites are specified to subscribe to data copies using the Replication Command Language (RCL). If an update is issued at a replica site, it is forwarded to the master site (write through) and then indirectly back to the original site. This is done by the Log Transfer Manager (LTM) that reads the log of the SQL Server to detect changes to replicated data. Every site that has a data copy must have an LTM. If such a change is detected the local Replication Server is contacted which in turn contacts the remote Replication Server of the site containing the master copy. Communication between the sites can be performed synchronously by invoking remote stored procedures or asynchronously through special “replicated” stored procedures at the remote

site. With asynchronous communication the procedure and its arguments are transferred asynchronously to the server with the master copy to be executed there. It is also possible to replicate horizontal fragments of a master copy to several different sites.

It is possible to implement a foreign data source by implementing your own customized Replication Server that detects changes to the foreign data sources and forwards them to sites containing copies, i.e. references to the foreign data source. Sybase thus supports the push model for propagating changes from foreign data sources. Since it is possible to define triggers on replicated tables, this technique makes it possible to use triggers to monitor changes of foreign data sources as well. To support updates of foreign data sources a special LTM has to be implemented that captures updates from the log and sends them in an acceptable format to the customized Replication Server.

IBM's DB2 Copy Management and the DataPropagator products

In IBM's DB2 Copy Management is based on the DataPropagator products. The DataPropagator Relational (DPROP/R) can propagate changes to tables between DB2 databases. DPROP/R works by capturing log records directly from the log buffer area. From this raw log data, logical log records are reconstructed that contain details on how the data has changed. DROP/R refers to control tables to determine which tables have been registered for capturing. Each registered table has a *changed data table* into which the changes are stored. When a user subscribes to a *registered table copy* an Apply Program (AP) must satisfy the subscription. The AP can run at the primary site or at any of the remote sites. At the primary site, the AP creates a *consistent changed data table* by using the changed data and appending four columns to each row containing transaction information (such as time of commitment, user ID, transaction ID) to provide time slices of the consistent data. The AP can operate in refresh or update mode. In the refresh mode it takes the complete source data copy and copies it to one or more targets. In the update mode the AP only captures and copies changes to the source data copy. This is useful if the tables are large since it limits unnecessary communication between sites. The AP can also handle changes to aggregate data.

Foreign data sources can be implemented with the DataPropagator Non-Relational (DPROP/NR) product. Using this the users can provide any data they wish as a consistent change table which will then be propagated by the AP. DPROP/NR will extract data from other data sources and format it to be loaded into a consistent changed table. DPROP/NR thus makes it possible to support foreign data sources with the push model for propagating changes. Monitoring of foreign data sources is also possible by using triggers. DPROP/NR will also capture updates from the IMS (Information Management System) log and put them into a consistent changed data table.

Oracle's Symmetric Replication

In Oracle's Symmetric Replication it is possible to define *snapshots* that are copies of master tables in some other server. The snapshots can be full copies of the master table or subsets of master table rows satisfying some value-based selection criteria. Oracle supports the pull model to propagate changes to all sites that have snapshots of the changed master table. Any changes to the master table since the last refresh will be propagated and applied to the snapshot at time-based intervals or on demand. Oracle also supports the push model for propagating changes. Like Sybase, Oracle follows a loose consistency model and provides the updates to the remote sites asynchronously. Unlike Sybase, Oracle does not scan the database log to detect updates; instead Oracle depends on triggers and asynchronous stored procedures for propagating changes. A trigger at the primary site fires on an update, insert, or delete. The trigger initiates execution of an asynchronous remote procedure call by submitting the request to a propagation queue. The requests are then forwarded to remote systems for execution within separate transactions. Asynchronous RPC transactions are executed on each remote system in the same order as they were committed to the local propagation queue.

Foreign data sources can be implemented with the Symmetric Replication by defining them as snapshot masters for read-only snapshots in Oracle. The Oracle server can then pull data at regular intervals or on demand. The push model can be implemented by having the data sources supported with asynchronous stored procedures that are called when the foreign data source changes and which store the changes in a local table (either just the changes or the whole table). Oracle will then support monitoring the foreign data sources through triggers. Updates of foreign data sources are not supported by Oracle's Symmetric Replication since snapshots are read-only.

9.2.2 Illustra/Informix DataBlade Modules

In Illustra [118], which has now become the Informix Universal Server [78], the concept of DataBlade modules has been introduced as a way to integrate new data structures into the DBMS. For example, a DataBlade module exist for storing time series in special data structures with various operations supported on the time series. Users can also create their own DataBlade modules that can be linked into the DBMS. This is supported by an extensible query optimizer that supports optimizing queries that access new data structures.

The system also supports SQL3 [90] which is a standardization of SQL with Object Relational extensions. SQL3 supports defining abstract data types and functions which provides some support for integrating foreign data sources in the database schema. Informix Universal Server supports asynchronous (non-blocking) I/O which is important for supporting access to foreign data sources without blocking the whole server. The DataBlade concept does not, however, include any way of signalling changes to data defined in the DataBlade so using SQL3 triggers to monitor changes to data in the DataBlade data structures is not supported.

Oracle and IBM/DB2 have announced similar functionality in future

releases, but it is unclear how they will compare with the DataBlade concept.

9.3 Accessing Foreign Data Sources

9.3.1 Foreign Tables/Foreign Objects and Attributes/ Foreign Functions

Foreign data sources can be introduced into a database in many ways. The most natural way is to introduce them into the database schema as if they represent stored data, i.e. as relational tables in a RDBMS, as objects with attribute values in an OODBMS, or as types and functions in an ORDBMS. This makes it possible to support iterative design techniques during the design of large systems where foreign data sources not yet available can be simulated by locally stored data.

To support foreign tables/attributes/functions the whole database must have been designed to be extensible. The query optimizer must have an extensible cost model to support optimizations of query plans that include accesses to foreign data sources. When discussing internal and foreign tables/attributes/functions we hereafter use the names *internal functions* and *foreign functions* for short.

9.3.2 Optimization of Queries that Access Foreign Data Sources

Queries and ECA-rules that reference foreign data sources must be optimized with the goal of minimizing all unnecessary communication. Accessing foreign data sources is generally more expensive than accessing local data, at least when indexes are available on the local data, but linear scans can sometimes make access of foreign data sources cheaper in comparison. Different foreign data sources might have different access costs, so the optimizer must not only have the information to compare local and foreign data access, but also in which order it should access different foreign data sources. If foreign data is stored locally in special data structures, the optimizer must also be aware of these and if it is possible to utilize any indexes when accessing them [118]. In some cases it might be possible to access the same the foreign data from different foreign data sources, the optimizer must then be able to make this decision. This is sometimes referred to as *performance polymorphism* [95].

9.3.3 Interrupting the Database

When an internal function changes, it can be monitored by change detection events. If a foreign function is to be monitored, then the changes to it have to be presented as events to the ADBMS as well. Changes to internal functions are usually related to logging, but changes to foreign functions are usually not logged (rolling back a transaction can not change the read values of a sensor).

An ideal scenario would be to let the external data source inform the DBMS

when it has changed by using interrupts. In most cases this would not cause too much extra load on the DBMS. We refer to this as a *push model* of communication where the foreign data sources pushes changes into the DBMS. If the push model is implemented with just OS signals to interrupt the DBMS, there is no support for transferring any data. In this case, the foreign data source can transfer data through shared memory (if this is supported by the OS) or by having the DBMS poll the foreign data source for the actual data when it receives an interrupt. This can be done by letting the interrupt handler of the DBMS schedule a background transaction that fetches the data. Protocols such as RPC and CORBA (see section 9.5.3) support the push model with transfer of data.

9.3.4 Polling Data Sources

In some cases the foreign data source has no support for interrupting the DBMS or the DBMS does not support being interrupted. In such cases the data source has to be polled by the DBMS. Another case is when a data source changes so frequently that interrupting the DBMS every time it changes would congest the DBMS. If the DBMS can poll a foreign data source, it can decide itself when it is time to check whether it has changed. We refer to this as the *pull model* of communication where the DBMS pulls changes from the foreign data source. Of course, when a foreign data source is pulled the new values will usually be sent to the DBMS to signal that it has changed. If it has not changed, an empty answer can be returned. A perhaps better scheme is to always return the latest value together with a timestamp that reflects the time when the foreign data source last changed. The database can then compare this with the timestamp of the previously read value to determine if the foreign data source has really changed. Foreign data that represents some alarm where the presence of an alarm is detected, but where the non-presence is not, will be considered as changed whenever a new alarm is detected. All protocols discussed in section 9.5.3 support the pull model.

9.3.5 Synchronous versus Asynchronous I/O

When a data source is polled it is important that this operation does not cause the DBMS to hang. When the database issues a read request to a foreign data source, it must still be available (both through standard interfaces and for interrupts from other data sources). In some cases it might even be desirable to support simultaneous polling of several data sources, e.g. when long communication delays are involved. To support this behaviour asynchronous (non-blocking) I/O is a must. The reading operation of a foreign data source should, however, appear as synchronous to the transaction that performs it. The transaction scheduler of the DBMS can suspend transactions while they are waiting for asynchronous replies from foreign data sources.

9.3.6 The Agenda

To support the polling of foreign data sources the DBMS must be able to schedule read requests. The DBMS must support cyclic polling in the background with different interval times for different foreign data sources as well as sporadic polling performed by users and applications. The actual times for polling should be determined from parameters defined for specific foreign data sources and from temporal event specifications in the active rules that monitor the foreign data sources. The *agenda* is a function that could support such behaviour.

It could be implemented using the UNIX *cron* daemon with its built in scheduling queue or by using the system timer and having the DBMS maintain its own scheduling queue or by reading the system clock at certain intervals and having the DBMS maintain both its own timer and scheduling queue. Which implementation scheme is the best depends on portability requirements, i.e. a system less dependent on OS support, and on efficiency requirements. The cron daemon maintains a file of the scheduled activities and is probably too slow for a DBMS with a high load on the agenda mechanism. Maintaining the scheduling queue in the DBMS has the advantages that the DBMS can better control the actual scheduling and is less dependent on the OS scheduler. It can cause extra overhead on the load of the DBMS, but will probably be more efficient than having the agenda mechanism outside of the DBMS.

9.4 Monitoring Foreign Data Sources

9.4.1 Foreign Events

To fit the ECA-model, foreign data sources will have to be defined through external events that signal a change to a foreign data source. We call these *foreign events*. Foreign events should always be defined together with a foreign data source. Introducing external events without actual data sources makes it possible to use the database rules for event programming (i.e. using just EA-rules) that perhaps is better supported in dedicated programming languages. In fact, it is our view that events should always signal change of some data (insert, delete, or update of data). This may involve changes to stored data, data stored in tables, stored object attributes, or stored functions. It may also involve changes to the schema. Changes to foreign data sources can be presented through foreign events as discussed here. Even temporal events can be viewed in this way; a temporal event signals a change of the foreign data source *time*. The only events that perhaps do not fit this model are events signalling access of data and signalling of the beginning and end of transactions (unless we regard the log with log records as changing data as well). By storing changes to foreign data sources in Δ-sets these can be used in event propagation and change propagation for partial differencing as presented in chapter 6.

A rule that monitors a foreign data source could look like:

on added/removed/updated(<foreign-function-call>) | <foreign-event-name>
when <predicate-expression>
do <procedure-expression>

The foreign data source can be monitored through the foreign events of a foreign function that represents added/removed/updated data or through some specially named foreign events. The condition here is a query that can access the value of the foreign data source, either through data passed from the event part or by directly accessing the data source (see section 9.3). Other foreign data sources not referenced in the event part as well as local data should, of course, also be accessible. The action can access foreign data sources as well and perhaps also update them (see section 9.4.2).

The external events can be raised directly by the foreign data source or by the ADBMS itself when it detects a change to a data source. Having data sources raising events themselves requires an interrupt procedure where the data source interrupts the ADBMS and transfers the necessary information (data) to support raising of the event. Having the ADBMS detecting changes to foreign data sources themselves requires a polling procedure where the database regularly polls the data source to detect changes. This can be done by having periodic temporal events that trigger reading of the data source. This can be done by ECA-rules, but is probably more efficient to hard-code into a special low-level agenda mechanism. Polling data sources is usually less efficient than letting them interrupt the ADBMS, but might be necessary if the data sources (or the ADBMS) lacks this capability. Polling the data sources might also be necessary if they are referenced in an ECA-rule containing temporal events [24][64][98][134]. Temporal events and temporal rules were also discussed in chapter 8.

For example, if we want to monitor a foreign function or foreign event every 10th second[1]:

on added/removed/updated(<foreign-function-call>) | <foreign-event-name> **and every** 10 seconds
when <predicate-expression>
do <procedure-expression>

The same mechanism as is used for background polling of the foreign data source could be used here but with a modified polling interval. The polling interval would have to be the minimum of all the temporal events referenced together with the foreign event and the background polling interval that has been specified for the foreign data source. This should be a task for the agenda (see section 9.3.6).

A rule such the one above probably requires immediate rule processing to be effective, unless transactions are shorter than 10 seconds in which case deferred rule processing might still be acceptable.

An alternative solution can be to extend the rule contexts in AMOS to sup-

1. Temporal events are currently not supported in AMOS, but can instead be introduced as foreign events that are raised periodically.

port contexts that are processed at certain time intervals. Any rule activated into such a rule context will be periodically checked. If several rules share the same interval specifications, then a periodically checked rule context is probably a better solution. If most rules have different interval specifications, they have to be processed separately and probably with an immediate coupling mode.

If we want to delay the rule condition check until a period after a certain event has been detected, we might want an ECA-rule like[1]:

on <event-type-specification> **after** 10 seconds
when <predicate-expression>
do <procedure-expression>

Another example can be a CA-rule where a check is done if a rule condition holds for a period of at least 10 seconds:

when <predicate-expression> **holds_for** 10 seconds
do <procedure-expression>

This rule will need a similar monitoring interval as for the previous rule, but with a check if the rule condition is true both at the beginning and at the end of a 10 second interval. Both of the last two rules probably need immediate rule processing.

These kind of rules will require modification of the rule execution presented in chapter 7 since it is based on simultaneous execution of events, conditions, and (after conflict resolution) the action. One possible solution could be to separate these kind of rules into one immediate EA-rule that, when being executed, schedules (with the agenda) a CA- or ECA-rule (in a special rule context) to be checked after a certain time period.

9.4.2 Updating Foreign Data Sources

If foreign functions are to be indistinguishable from internal functions they have to be updatable as well. Since an internal function just represent stored data, it can easily be changed. A foreign function that represents a value that is stored outside the database can perhaps be changed. A foreign function, however, can also represent information which can not be directly changed. In some applications it might be desirable to define the update of a foreign function to be an operation that indirectly changes its value. If the foreign function represents a sensor, its update might be defined to operate an actuator that indirectly changes the sensor value, e.g. changing the value of a thermometer by operating a heater or changing detected congestion in a network by rerouting messages over other links.

Allowing update operations of foreign data sources can be seen as closing the control loop and allowing an ADBMS to indirectly participate in all parts of the execution cycle of a control application. Many control applications are,

1. Currently not supported in AMOS.

however, real-time applications and might put demands on the efficiency and predictability of the ADBMS that it might not meet. Closing the control loop might be desirable in some applications, but should not be seen as letting the ADBMS take the role of dedicated real-time control systems. With more support for real-time behaviour in ADBMSs [101], the number of applications where the ADBMS can participate in the whole control loop might increase.

9.4.3 Callbacks and Notifications

When the ADBMS triggers a rule due to the change of a foreign data source, the action might be to update the foreign data source. Another response might be to issue a *callback* or a *notification* to an external application. Update procedures of foreign data sources will often be defined to issue callbacks to an external application that indirectly updates the foreign data source. The callback can be represented by a general callback mechanism or by dedicated functions that each communicate with a specific external application. A notification could be to open a notification window to signal a certain situation or to sound an alarm. If the rule is some kind of constraint rule, the callback can be to an external function that resolves a constraint violation, in the case where the ADBMS cannot resolve it directly itself.

9.4.4 Storing Foreign Data

Even though foreign functions represent foreign data, it might still be desirable to store the data in the database. This could be for persistency reasons, i.e. to log all data for later analysis, or for support of temporal queries over time series representing the historic changes of the foreign data sources. Another reason might be to avoid unnecessary polling of foreign data sources by storing the latest value(s) semi-permanently in a cache. By timestamping data which has been read, the latest reading of a data source can be checked to see if it is good enough to use instead of polling the foreign data source for a new value. This operation is usually much cheaper than actually polling the data source. The extent to how old values we allow depends on the actual foreign data source and the monitoring situation. These decisions will usually be made at run-time. In a *performance polymorphic query optimizer* [95] the decision of how to access a data source can sometimes be made at compile time (during query optimization).

9.4.5 Storing Foreign Events

In an ADBMS the events are usually stored, either in the transaction event log or in dedicated data structures as the time series used for Δ-sets in AMOS. The same data structures can be used for storing foreign events, but since the foreign events can occur in the background they are usually not related to one particular transaction so logging them in a transaction event log is not a good solution. If events are stored in some special data structure, such as Δ-sets that are used for propagating events to incrementally calculate complex events (events defined in terms of other events),

these could be used for storing foreign events as well. If we want to support complex events defined in terms of foreign events this could then be done in a similar manner as for internal events. Foreign data sources might, however, change very rapidly and this might give rise to problems such as overflowing the database if all the related foreign events are stored over long periods of time. If we want to combine temporal events with foreign events, we also need to timestamp when they occurred (this might be the case for internal events as well).

9.4.6 Time Series and Time Windows

To support temporal queries and temporal ECA-rules that reference foreign data sources it might be necessary to store foreign data in time series. Since a data source might change very rapidly it is crucial that the time series can be updated without too much overhead. Storing the time series directly in a tree structure that is indexed by time will cause to much overhead since the tree will have to be re-balanced as the time series grow. Still it is necessary to support fast access of particular time points in the time series without linearly scanning them. This can be done by special index data structures or by using computational indexes [45]. It might also be necessary to limit the size of time series and to discard old values when they become full. This can be done automatically by defining sliding time series, *time windows*. See chapter 8 for more discussions on time series.

9.5 Implementation Issues

9.5.1 Connecting Foreign Data Sources to a Database

Basically foreign data sources can be connected in two different ways to the database. The data source can be tightly connected by linking the code of the data source with that of the DBMS. This requires, of course, that the DBMS supports this, which is not the case with most commercial DBMS products. A more common model is to connect the data source through some client/server interface that supports foreign data sources. Most commercial databases do not support this either, but hopefully they will in the future. Some commercial DBMSs have replication servers that can be used to connect foreign data sources, as discussed in section 9.2.1. As discussed in section 9.3, the actual monitoring of changes to the foreign data sources can be done in three different ways:

- The data source interrupts the DBMS when it has changed and writes its new value directly into the database.
- The data source interrupts the DBMS when it has changed and the DBMS polls for the new value.
- The DBMS polls the data source at periodical intervals or when needed.

Combinations of these techniques can be used on the same foreign data source depending on what is needed to support the application tasks. For example, in a combination of interrupt driven monitoring and polling, the read operation can look at the timestamp of the latest received data to decide whether to actually poll the data source. How the foreign data sources are to be connected has to be declared in the database schema. This includes information about which models of communication should be used, i.e. push- or pull-based. If the pull model of communication is defined, then the polling intervals will have to be specified. The actual polling intervals will be decided dynamically by all the temporal events that are specified in the rules that monitor the actual foreign data sources. Actual communication protocols will also have to be specified, as well as any translations that have to be performed on the data. Information needed by the optimizer to optimize queries that involve foreign data source access has to be specified as well.

9.5.2 Declaring Foreign Data Sources in the Database Schema

When foreign data sources are declared in the database schema, they should be declared in terms of the standard way of declaring data, i.e. as special tables, objects and attributes, or as types and functions. Several pieces of information have to be provided with these declarations:

- The name of the foreign data source.

- Data type declarations.
  Information about what data type(s) will be returned from the foreign data source and, if it can be parameterized, the types of the parameters.

- Modes of communication (push, pull, or both).
  If the pull model is specified then information about when it should be polled, e.g. by specifying the time interval.

- Foreign event specifications.
  Information about what foreign event should be raised to signal a change of the foreign data source and how the change will be detected. If the push model is specified, then information about what interrupt will be used has to be specified. If the pull model is specified, then information has to be specified as to how a change is detected in polled data, e.g. by a new value or a newer timestamp.

- Update information.
  If the foreign data source can be updated, it has to be specified how. This could be done by specifying special user defined update procedures. If the updates are to be logged, undo operations will have to be specified as user-defined procedures as well.

- Storage information.

If the foreign data source is to be stored permanently or semi-permanently, this has to be specified along with information about what data structures should be used, and whether there are any indexes on the parameters and return values.

- Optimization information.
  To help the optimizer, information about costs of accessing the foreign data source, e.g., communication costs, translation costs, and the cost of accessing any special data structures have to be specified. If the same foreign data can be accessed from other foreign data sources, this has to be specified as well.

- Actual communication protocols that will be used.
  The specified protocols must match the specified communication modes as well as foreign event specification, i.e. if the specified protocol can provide them. If any conversions to the data and parameters are needed, this has to be specified as well.

- Physical connection data.
  Information about how the foreign data source is physically connected, e.g. what IP addresses, port numbers, file descriptor numbers, or on what physical addresses in memory data will be written.

9.5.3 Standard Protocols

Low level protocols such as RS-232, X.25, X.29, dedicated real-time protocols such as real-time Ethernet, instrumentation buses such as Fieldbus, and telecommunication protocols such as CCS7, ISDN and ATM can all be used for connecting foreign data sources to a database, but are not discussed here. Software protocols like TCP/IP (discussed below) can be used on top of hardware implemented transport protocols like ATM (discussed in section 2.5) and SCI (discussed below).

TCP/IP, SMTP

TCP/IP [115] is the packet-based protocol for communicating over the Internet, but can be used on local communication networks, e.g. a local Ethernet network, as well. It is based on a global addressing scheme where all sites are uniquely identified by global addresses. TCP/IP provides reliable two-way connections, e.g. by *sockets*, that can be used for both pushing and pulling data. Sockets can thus provide both synchronous and asynchronous communication between databases and foreign data sources. Pushing the data requires a local interrupt daemon that informs the DBMS that new data has arrived.

SMTP [115] is the TCP/IP electronic mail protocol. Providing SMTP interfaces to DBMSs makes it possible to use database technology to better handle large amounts of e-mail. For example, the problem of mass copying broadcasted mails as is done today might be avoided by storing the mails in one or a few databases and accessing them instead of copying data. Better search facilities would also be provided by the full use of database query languages to

access the mailboxes. SMTP is basically a push protocol where mails are sent out and the recipients are notified by the mail server. It is possible to pull information from most SMTP servers by monitoring changes to special files as is done by the *biff* program which can inform when new mail has arrived and the *finger* program which checks if and when a user has read his/her mail.

RPC

In Remote Procedure Calls (RPC) [12][115] the client can call remote procedures in the server as if they are local procedures. RPC packages are usually built on top of the TCP/IP protocol, but can also be built on other protocols as well. The RPC package will provide the client with procedure stubs, *client stubs*, that actually calls the server. The RPC provides the server with corresponding *server stubs* that receive calls from the client stubs. Calls from client stubs are packaged into a network message containing the procedure arguments. The server stubs unpack the network messages and send the procedure arguments to the actual server function. When the server function returns to the server stub, the return values are packed in a network message which is sent back to the client stub. The client stub finally returns the result to the caller of the RPC to make it appear as if it was a local procedure call.

RPC based communication is common for connecting databases over a network and can be used as a synchronous, low-level connection protocol between databases and foreign data sources.

File Access, NFS, FTP

Foreign data sources defined as external files are very common. If these are accessible through remotely mounted file systems, they can be accessed with protocols such as NFS (Network File System) [115]. If the files are not on remotely mounted file systems, they can be accessed with protocols such as FTP (File Transfer Protocol) [115]. The data accessed in files usually have to be translated into data readable by the DBMS.

HTTP, CGI

HTTP (Hyper Text Transfer Protocol) [46] is a standard protocol for distributed hypermedia applications on the World-Wide Web (WWW) and is used by Web browsers, for example. HTTP is a pulling protocol where the foreign data sources, e.g. WWW-pages will have to be fetched from WWW servers and then be scanned by the DBMS. Using a DBMS for accessing the WWW can help to better search the vast amount of information available on the WWW. Web crawlers and search engines are examples of the use of DBMSs for accessing the WWW.

The Common Gateway Interface (CGI) is a standard for interfacing external applications with information servers, such as HTTP or WWW servers. A plain HTML (Hyper Text Markup Language) document that the WWW daemon retrieves is static, which means it exists in a constant state: a text file that does not change. A CGI program, on the other hand, is executed in real-time, so that

it can output dynamic information. If one wants to connect a DBMS to the WWW where queries can be put to the DBMS and where the answers are dynamically created, then a CGI gateway can be used. The interface will be a CGI program that the WWW daemon (a local process interacting with the WWW server) can execute to transmit information to the DBMS, and receive the results back again and display them to the client. Vendor-specific extensions to CGI have been provided such as NSAPI (Netscape Server Application Programming Interface) and ISAPI (Internet Server Application Interface).

CORBA

The Common Object Request Broker Architecture (CORBA) [135] is quickly becoming the de facto standard for achieving interoperability between Object-Oriented systems. CORBA is defined within the Object Management Architecture (OMA) by the Object Management Group (OMG). OMA defines the Object Request Broker (ORB) as a common communication bus for objects and where CORBA is an architecture for distributed objects. CORBA defines:

- Exportable object identifiers
- Static and dynamic invocation interfaces
- Interaction models

CORBA makes it possible to export objects as object references from one system to another. A DBMS with a CORBA-interface could thus reference foreign data sources through CORBA-objects exported from some application with a CORBA interface.

CORBA defines both static and dynamic invocation interfaces. In the static invocation the interface is determined at compile-time and is presented to client code using code stubs defined in the OMG Interface Definition Language (IDL). In a DBMS foreign data sources can be defined by IDL stubs that defines the objects of the foreign data source (this can be used with an OODBMS where the database schema usually is static anyway). In the dynamic invocation interface the clients can construct and issue requests whose signatures are not known until run-time. The dynamic interface can be used by a DBMS to connect foreign data sources dynamically and where it is not possible to compile and link the database schema with static code (which is the case with most RDBMSs).

The CORBA interaction models are based on RPC, i.e. synchronous invocations. CORBA also supports asynchronous interaction by allowing a client to continue without waiting for the result of a request. In the dynamic invocation interface CORBA also supports *deferred synchronous* interactions where the client can receive a response some time after issuing a request and after having done something else instead of just waiting for the response. Supporting both synchronous and asynchronous interaction is important for defining different kinds of foreign data source interfaces.

Within OMA an Object Event Notification Service has been defined that

supports notification of events to interested objects. It defines objects to have *supplier* roles (produces data) and *consumer* roles (consumes data). The event data are communicated between suppliers and consumers by issuing standard CORBA requests. Two models for communicating events are defined. A *push model* where the supplier of events initiate the transfer of event data to the consumer(s). This is similar to the push model defined in this chapter where a foreign data source (the producer) interrupts a DBMS (the consumer) to signal that it has changed. In the *pull model* a consumer requests event data from a supplier. This is similar to the pull model where a DBMS polls a foreign data source to detect if it has changed.

The service also defines *event channels* that are intervening objects that allow producers to communicate with multiple consumers asynchronously. Different event channels need to be implemented to provide push or pull models or a combination of these.

Another service within OMA is the Object Transaction Service (OTS) which provides ACID-transactions. More specifically, OTS supports development of transactional object classes. OTS defines both flat and nested transactions. In a CORBA interface to a DBMS these have to be mapped to the transactions provided by the DBMS. If the DBMS has defined *coupling modes* for active rule execution, e.g immediate, deferred, and decoupled, that are related to the transactions, these could be defined to relate to the CORBA transactions.

SQL/CLI

SQL/CLI [128] is a standard client/server interface for applications interacting with a relational (SQL) DBMS. It is based on that the application (the client) only wants to access and store data permanently within the DBMS (the server). If the application is to act as a foreign data source to the DBMS, this can only be achieved by having the application updating a stored table in the database. This can be seen as push-based communication where the data is stored permanently or semi-permanently (the foreign data source will have to delete old data itself) in the DBMS. SQL/CLI also defines how the application can specify how data from a table should be translated to fit the data format of the application language. This can be useful for defining translation of data from external data sources so that it conforms with the data formats of the DBMS. SQL/CLI is based on synchronous communication and where all operations are performed (and logged) inside a transaction. SQL/CLI as it is defined today is not sufficient as a general foreign data source interface.

OLE DB/ODBC

OLE DB [14] is a new set of interfaces being developed by Microsoft. It is an extension of OLE/COM (Object Linking and Embedding/ Component Object Model) with better support for database integration. OLE DB supports representing foreign data in a tabular format to make it accessible in SQL. Transparent access of the foreign data sources is supported. Notifications can be defined to monitor changes to a foreign

data source. Updates can be provided by special method calls. OLE DB is defined in OLE COM and is thus mainly for integrating DBMSs and applications within the Windows and WindowsNT platforms. ODBC (Open Database Connectivity) is a general protocol that supports full SQL and SQL/CLI can be seen as a subset of ODBC. Compatibility between OLE DB and ODBC is provided through an OLE DB library. It is also possible to extend OLE DB with user defined communication protocols. OLE DB has support for adding new data structures through *data providers*, as long as they expose their data in a tabular format. Support for extensible optimizers is also possible by support for specifying *optimization goals*, e.g. limits on CPU time, memory utilization, I/O, or network messages.

Microsoft have also defined their own protocol, DCOM (Distributed Component Object Model), for distributed communication for integrating different resource managers such as SQL-servers and MTSs (Microsoft's Transaction Servers). DCOM is a competitor to CORBA. Microsoft also have a distributed message server, MSQS (Microsoft's Message Queue Server), which uses RPCs for transferring messages and DCOM to exchange control and management information among queue manager nodes.

Low level transport protocols (ATM and SCI)

TCP/IP can be defined on top of a protocol like ATM (fig. 9.3) where IP packages are fragmented into ATM packages (cells) at the source and then are assembled back into the IP package at the destination. This will make it possible to extend the Internet to run on future broad-band telecommunication networks. DBMSs for network applications can interconnect over the TCP/IP interface without bothering about over which networks the connections are being set-up. ATM ports in network devices may be assigned IP addresses which makes it possible to reference any network element using IP addresses.

A new standard, MPOA (Multi Protocol Over ATM), has been defined that allows ATM networks to better support *internetworking*, i.e. integrating local sub-networks with their own protocols through ATM networks. See section 2.5 for a more thorough discussion of ATM in telecommunication networks.

Another interesting transport protocol for interconnecting DBMSs and foreign data sources is the SCI (Scalable Coherent Interface) [70]. The SCI is a specification developed to provide high bandwidth and the ability to connect a large number of processors, memory, and I/O devices. These devices are connected via a point to point interconnect. The SCI interface is mostly used for fast mulitprocessor interconnection [124], but can also be used for connecting different peripherals to a computer system. The SCI standard provides:

- *Scalability*: The network performs well in systems scaling from a few to a large number of processors.
- *Coherence*: The distribution is transparent through a distributed shared memory.
- Standard *Interface*: The provided interface is not restricted to a particular principle or technology and can be provided by many different vendors.

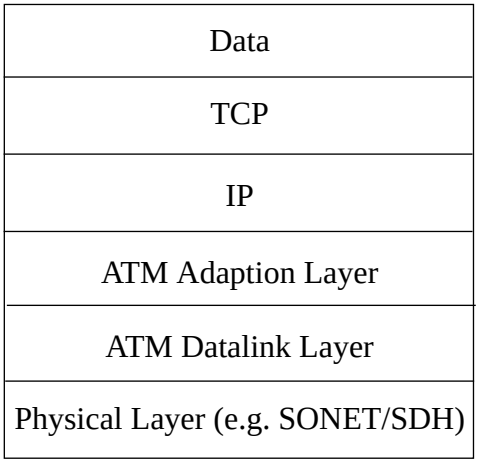


Figure 9.3: Running TCP/IP on top of ATM

SCI provides a directory-based cache coherence protocol for the processors or peripherals (nodes) to exchange information. The directory provides a shared memory between the SCI connected nodes through cache memories in each node. Each node communicates by reading and writing into the cache memories. The reading and writing is synchronized using locks (basically semaphores). The SCI standard supports up to 64K nodes and provides a raw point-to-point throughput of 1 Gbyte/sec. Since SCI is based on shared memory all the involved nodes must store the data in the same manner, i.e. when considering significant bits (Little or Big Endian architectures) and how the data is interpreted (how different data types are encoded). TCP/IP can also be set-up on top of SCI to provide more transparency, but at the cost of assembling/disassembling data into IP packages and translating data instead of reading it directly in the shared memory (data type encoding will still have to be performed at both ends).

SNMP/MIBs

The Simple Network Management Protocol (SNMP) [1 14] is a protocol for Internet network management services. In SNMP there is a separation between the management system and the managed device (network element). SNMP basically supports four dif ferent operations. Three operations are specified from the management system to the managed device:

1. Get - Retrieve one element of management data from an IP addressable device.
2. Get next - Get the next element of data from that device.
3. Set - Modify an element of management data in that IP addressable device.

One operation is defined from the managed device to the management system:

4. Trap - Tell the management system that something has happened. Can be used for sending alarms and for monitoring the status of the network device.

Within SNMP the Management Information Base (MIB) is a group of standards for different network devices that should be accessible through SNMP. An ATM MIB [3] is defined for managing network elements in an ATM-network. In the NSM (Network Services Monitoring) MIB generic attributes have been defined for managing network applications. In the RDBMS MIB [16] most RDBMS vendors have agreed on how to access and manage RDBMSs over a network. The RDBMSs are defined as servers that can be:

- 1 database : 1 server
- 1 database many servers
- many databases : 1 server
- many databases : many servers

The RDBMS MIB sees the servers as a collection of tables and has nine MIB-specific tables for managing the servers:

- databases installed on the host/system
- actively opened databases
- database configuration parameters
- database limited resources
- database servers installed in a system
- active database servers
- configuration parameters
- server limited resources
- relation of servers and databases on host

A special SNMP Replication MIB has also been defined for managing replicated data in a distributed DBMS. Other proposed MIBs are MADMAN (Mail And Directory MANagement) MIB, WWW MIB (extension to NSM MIB), and an HTTP MIB.

If an ADBMS is to be used for network management it has to support standard SNMP MIBs. If network elements are introduced as foreign data sources the monitoring functionality of SNMP MIBs (the Trap operation) can be introduced as foreign events that can be monitored through ECA-rules.

9.5.4 A Foreign Data Source Protocol

The above protocols are not enough for connecting foreign data sources to a database. Some of them have the notion of events, such as CORBA and OLE DB, but this need to be specified more exactly in order to support real interoperability between foreign data sources and databases. OLE DB actually has support for foreign data sources, but is platform- or vendor-specific. In many cases protocols such as CORBA are too heavy to use between simple data sources and databases. Many data sources can have simple non-OO data models, perhaps just a couple of sensors defined as simple foreign functions returning real valued data. In those cases it would be desirable to have a minimal protocol for minimal overhead (such as SNMP MIBs). Such a limited protocol could be sometimes mapped to standards such as CORBA or general protocols such as OLE DB to provide general foreign data source access. Protocols such as SMTP have the notion of events defined as arriving mail to a users mailbox and should be possible to map to a more general protocol. It should be possible to map HTTP to a general protocol as well.

9.5.5 Connecting Foreign Databases

As systems and business platforms are becoming more heterogeneous, there is a growing need for database interoperability. One solution is to connect different databases though mediating software. Products are available on the market and there are several research platforms. None of these support interoperabillity from an active database standpoint. There is no coherent way to connect different active databases and there are no products that support a coherent way of specifying active rules that refer to several heterogeneous databases. In [119] an extension of CORBA is defined for interconnecting different heterogeneous DBMSs using rules.

9.5.6 Mediating DBMSs

Mediating DBMSs [41][131] work as a glue between different databases and can provide translations between different data models such as relational to Object Oriented [42]. Mediating databases can be used in connecting heterogeneous databases as foreign data sources. Support for accessing remote databases and updating them are available in protocols like SQL/CLI, ODBC, CORBA, and to a limited extent in commercial replication servers (see section 9.2.1).

9.5.7 Exporting Events

In order to provide monitoring capability of foreign data sources represented as other databases, the communication protocols must be provided for exporting events that make this possible. CORBA provides events that could be used for this purpose. OLE DB notifications could also be used for this. Techniques used in commercial replication servers (section 9.2.1) also provides means for exporting events, i.e. information about changed tables, but these are specific for each vendor.

9.5.8 Compiling Down Triggers

If foreign data sources that represent tables in other databases are referenced in ECA-rules we must be informed when these tables change. Instead of exporting all events of referenced tables it might be possible to compile down triggers into the remote database (provided it supports triggers). This technique is used in the Oracle replication server for detecting changes to replicated tables (see section 9.2.1). If several foreign data sources in the same remote database are referenced in one ECA-rule, e.g. in a conjunctive event specification, this technique might be very beneficial since we only export the events if they really might trigger the rule. Another way to approach this is to view this as exporting more complex events involving several data sources.

9.6 Foreign Data Sources in AMOS

Foreign data sources as presented here are not yet implemented in the AMOS system. AMOS does support the concept of foreign (external) events, but these have to be specified manually by registering a new event type by name with the event manager along with an event function that stores the event data. The new event can be manually raised by:

raise(<event-name>, <event-data>);

This operation raises the event and associates the transaction time and the specified event-data with the event. As can be seen in the event specification presented in section 3.8, foreign events can be specified by naming the event specifically:

event-type-specification ::=
added(*function-call*) |
removed(*function-call*) |
updated(*function-call*) |
created(*variable-name*) |
deleted(*variable-name*) |
foreign-event-name |
event-type-specification **and** *event-type-specification* |
event-type-specification **or** *event-type-specification* |
event-type-specification **before** *event-type-specification* |
event-type-specification **after** *event-type-specification*

A future extension of this will be to automatically create foreign events as part of the foreign function definitions. Note that if data of a foreign function can be added, removed, and updated, it can have events defined for it as well which can be monitored just as if it was a stored function.

Foreign data sources in AMOS are currently used for extending the DBMS with new data structures and operations (similar to the Informix DataBlade concept). A new data structure for storing time series has been implemented in

AMOS to support storing events monitored by active rules, this data structure is discussed in chapter 8. The AMOS foreign data source concept is being extended to support external data access as discussed in this chapter. This will sometimes include defining new data structures, but the focus this chapter is on access to external data. Foreign data sources in AMOS will be based on an extension of the foreign function concept in AMOS. A foreign function of a foreign data source could be declared as:

create function *function-name parameter-specification result-specification*
as foreign *foreign-data-source-access-specification*
[**set** *foreign-data-source-change-specification*]
[**add** *foreign-data-source-change-specification*]
[**remove** *foreign-data-source-change-specification*]
[**event** *foreign-event-name*]

foreign-data-source-access-specification ::=
[*implementation-specification*]
[**size** *integer*]
[**push** *interrupt-method* [*file-descriptor* | *address*]]
[**pull frequency** *time*]
[**transformations** *transformation-specification*]
[**costhint** *optimization-specification*]

foreign-data-source-change-specification ::=
function-call
[**event** *foreign-event-name*]

The foreign data source specification includes information such as the name of the actual function, access specification, set (update), add, or remove change specifications, and any specific event name specification. The access specification includes specific implementation information such as the actual communication protocol (if any), physical addresses, IP addresses, port numbers, and interrupt signals. The size defines if the foreign function should have a cache and of what size. Push and pull specifies access method and this must be matched by the actual communication protocol that has been specified. Transformations specify any transformations of the physical data when it is accessed such as rounding or transforming continuous values to discrete values (sampling) or transformations such as transforming discrete values to continuous (interpolation). The costhint specification is used for specifying different access patterns to allow inverse queries and also what costs are involved in accessing the foreign data source (see [82] for more details). The costhint specification can also be extended to support real-time costs as is defined in [95].

Different methods can be defined for updating, adding, or removing data of the foreign function (if supported at all). The change specification specifies

how changes to the foreign function will be signalled to support defining ECA-rules that monitor changes to the foreign function. This could register poll operations with the agenda that reads the foreign data at certain intervals and raises the event if a change is detected or an interrupt routine that raises the event when a specified interrupt occurs.

10 Conclusion

10.1 Summary

In this thesis some background work from the application areas of Computer Integrated Manufacturing (CIM) and Telecommunication Network Management (TNM) have been presented. These applications served as motivation for the functions of an Active Database Management System (ADBMS) that were presented in the thesis. The major contributions within the field of active database systems are:

- Identifying the need of ADBMSs through the *case studies* of CIM and TNM. In the application studies the requirements for efficient execution of rules with complex conditions and the need for transparent access of external data were identified.
- Using active rules for *monitoring* and *control* in CIM and TNM.
- Identifying the need for *mediators* in CIM and TNM.
- Defining an ADBMS architecture.
- Identifying the need for generalizing the architecture towards *active mediators*.
- Adding active rules to an Object-Relational DBMS.
- Integrating (E)CA-rules into a query language.
- Rule modularization by grouping rules into *rule contexts.*
- Efficient rule evaluation techniques based on *partial differencing.*
- Defining external data in a transparent manner through the concept of *foreign data sources*.
- Defining external events through the concept of *foreign events* of *foreign functions*.
- Work on introducing *time series* for storing event histories.
- Work on new *indexing techniques* for inverse queries over time series.

Most of the ideas presented here, such as CA- ECA-, EA-rules, partial differencing for efficient rule condition monitoring, time series for storing timestamped events in event logs, and rule contexts have been implemented in the AMOS ADBMS.

10.2 Future Work

Future work includes a full implementation of the rule system with support for the aspects presented here, for example, defining and monitoring foreign data sources, larger applications, incremental evaluation of aggregates, temporal events, rule conditions using temporal queries, and multidatabase rules.

13 References

[1] Abbott J. C.: Sets, Lattices and Boolean Algebra, *Allyn and Bacon*, 1969.

[2] Adelberg B., Kao B., and Garcia-Molina H.: Overview of the STanford Real-time Information Processor (STRIP), *SIGMOD Record*, Vol. 25, No. 1, March 1996.

[3] Ahmed M. and Tesink K.: Definitions of Managed Objects for ATM Management Version 8.0 using SMIv2, Network Working Group, Stand. Doc. RFC 1695, August 1994.

[4] Ahn I.: Database Issues in Telecommunications Network Management, *SIGMOD 5/94*, 1994.

[5] Alexander P. and Carpenter K.: ATM Net Management: A Status Report, Data Communications, Tech. Tutorials, Sept. 1995.

[6] Astrahan M. M., Blasgen M. W., Chamberlin D. D., Eswaran K. P., Grey J., Griffiths P. P., King W. F, Lorie R. A, Mc Jones P. R., Mehl J. W, Putzolu G. R., Traiger I. L., Wade B. W., and Watson V. : System R: Relational Approach to Database Management, *ACM Transactions on Database Systems*, Vol.1, No. 2, June 1976, Pages 97-137.

[7] Baekgaar L. and Mark L.: Incremental Computation of Nested Relational Query Expressions, *ACM Transactions on Database Systems*, Vol. 20, No. 2, June 1995, Pages 111-148.

[8] Baralis E., Ceri S., Fraternelli P., and Paraboschi S.: Support Environment for Active Rule Design, *Journal of Intelligent Information Systems*, 7, Pages 129-149, 1996.

[9] Baralis E. and Widom J.: Using Delta Relations to Optimize Condition Evaluation in Active Databases, *RIDS'95(Rules in Database Systems)*, Springer Lecture Notes in Computer Science, pp. 292-308, Athens, Greece, Sept., 1995.

[10] Beech D.: Collections of Objects in SQL3, in *the 19th International Conference on Very Large Databases* (VLDB'93), Dublin, Ireland, 1993, Pages 244-255.

[11] Berndtsson M. and Hansson J.: Issues in Active Real-Time Databases, Proceedings of Active and Real-Time Database Systems (ARTDB-95), Pages 142-157, Skövde 1995.

[12] Bernstein A. J. and Lewis P. M.: *Concurrency in Programming and Database Systems*, Jones and Bartlett Publishers, Inc. ISBN 0-86720-205-X, 1993.

[13] Blakeley J. A., Larson P-Å., and Tompa F.W.: Efficiently Updating Materializing Views, *ACM SIGMOD Conference*, Washington D.C., 1986, pp. 61-71.

[14] Blakeley J. A.: Data Access for the Masses through OLE DB, *Proceedings of ACM SIGMOD* International Conference on Management of Data, Montreal, June 1996, Pages 161-172.

[15] Blumenthal M. S.: Unpredictable Certainty: The Internet and the Information Infrastructure, *IEEE Computer*, Jan. 1997, Pages 50-56.

[16] Brower D. (ed.): Relational Database Management System (RDBMS) Management Information Base (MIB) using SMIv2, Network Working Group, Stand. Doc. RFC 1697, August 1994.

[17] Brownston L., Farrell R., Kant E., and Martin N.: *Programming Expert Systems in OPS5*, Addison-Wesley, 1985.

[18] Buchman A. P., Branding H., Kudrass T., and Zimmermann J.: REACH: a REal-time, ACtive and Heterogeneous mediator system, *IEEE Data Engineering Bulletin*, Vol. 15, No. 1-4, Dec. 1992, Pages 44-47.

[19] Bukhres O. A. and Elmagarmid A. K.: *Object-Oriented Multidatabase Systems - A Solution for Advanced Applications*, Prentice-Hall Inc., ISBN 0-13-103813-2, 1996.

[20] Cattell R. G. G: *The Object Database Standard: ODMG-93b Release 1,2*, Morgan Kaufmann Publishers Inc., 1994.

[21] Ceri S., Gottlib G., and Tanca L.: What You Always Wanted to Know About Datalog (And Never Dared to Ask), *IEEE Transactions on Knowledge and Data Engineering*, Vol. 1, No. 1, March 1989.

[22] Ceri S. and Widom J.: Deriving Production Rules for Incremental View Maintenance, In *Proceedings of the 17th VLDB Conference*, Brisbane, Queensland, Australia, Aug. 1990, Pages 577-589.

[23] Chakravarthy S., et. al.: HiPAC: A Research Project in Active Time-Constrained Database Management, *Xerox Advanced Information Technology*, Technical Report XAIT-89-02, Cambridge, MA, Aug. 1989.

[24] Chakravarthy S. and Mishra D.: An Event Specification Language (Snoop) for Active Databases and its Detection, *UF-CIS Technical Report, TR-91-23*, Sept. 1991.

[25] Chakravarthy S., Krishnaprasad V., Anwar E., and Kim S-K.: Composite Events for Active Databases: Semantics, Contexts and Detection, in *Proceedings of the 20th VLDB Conference*, Santiago, Chile, 1994, Pages 606-617.

[26] Chandra R. and Segev A.: Active Databases for Financial Applications, *RIDE ´94*, Houston, Febr., 1994, Pages 46-52.

[27] Chawathe S. S., Garcia-Molina H., and Widom J.: A Toolkit for Constraint Management in Heterogeneous Information Systems, *Proceedings of the Twelfth International Conference on Data Engineering*, New Orleans, 1996, Pages 56-65.

[28] Chimenti D., Gamboa R., and Krishnamurthy R.: Towards an Open Architecture for ℒDL, *Proceedings of the 15th VLDB Conference*, 1989, Pages 195-205.

[29] Dayal U., Blaustein B., Buchmann A., Chakravarthy, Hsu M., Ledin R., McCarthy D., Rosenthal A., and Sarin S.: The HiPAC Project: Combining Active Databases and Timing Constraints, *SIGMOD Record*, Vol. 17, No. 1, March 1988.

[30] Dayal U., Buchman A.P., and McCarthy D.R.: Rules are Objects too: A Knowledge Model for an Active, Object-Oriented Database System, *Proceedings of the 2nd International Workshop on Object-Oriented Database Systems*, Lecture Notes in Computer Science 334, Springer 1988.

[31] Dayal U. and McCarthy D., The Architecture of an Active Database Management System, in *Proceedings of the ACM SIGMOD Conference*, 1989, Pages 215-224.

[32] Dayal U., Hsu M., and Ladin R.: Organizing Long-Running Activities with Triggers and Transactions, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Atlantic City, May 1990.

[33] Dewan H. M., Ohsie D., Stolfo S. J., Wolfson O., and Da Silva S.: Incremental Database Rule Processing in PARADISER, *Journal of Intelligent Information Systems*, 1:2, 1992.

[34] Dewitt D.J., Katz R.H., Olken F., Shapiro L.D., Stonebraker M.R., and Wood D.: Implementation Techniques for Main Memory Database Systems, *SIGMOD Record*, Vol. 14, No. 2, 1984, Pages 1-8.

[35] Dittrich K. R., Gatziu S., and Geppert A.: The Active Database Management System Manifesto: A Rulebase of ADBMS Features, In *the Second International Workshop*

on Rules in Database Systems (RIDS'95), Athens, Greece, September 25-27, 1995, Springer Lecture Notes in Computer Science, ISBN 3-540-60365-4, Pages 119-130, 1995.

[36] Dong G. and Su J.: First-Order Incremental Evaluation of Datalog Queries, *Proceedings of the Fourth Intternational Workshop on Database Programming Languages - Object Models and Languages, Springer-Verlag,* Aug. 30, 1993, pp. 295-308.

[37] Dyreson C.E. and Snodgrass R.T: Efficient Timestamp Input Output, *Technical Report of the Department of Computer Science, University Arizona*, TR 93-01, Febr., 1993.

[38] Eich M. H. (ed.): Main-Memory Databases: Current and Future Research Issues (foreword), Special section on main-memory databases, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 6, December 1992.

[39] Elmasri R. and Navathe S. B.: *Fundamentals of Database Systems, Second Edition*, The Benjamin/Cummings Publishing Company, Inc., ISBN 0-8053-1753-8, 1994.

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

[41] Fahl G., Risch T., and Sköld M.: AMOS - An Architecture for Active Mediators, *International Workshop on Next Generation Information Technologies and Systems (NGITS '93)*, Haifa, Israel, June 1993, Pages 47-53, (in this thesis as Paper VI).

[42] Fahl G. and Risch T.: Query Processing Over Object Views of Relational Data, to appear in *the VLDB Journal*, 1997.

[43] Falkenroth E., Törne A., and Risch T.: Using an Embedded Active Database in a Control System Architecture, in *the 2nd International Conference on Applications of Databases*, San Jose, CA, USA, Dec. 1995.

[44] Falkenroth E.: Data Management in Control Applications - A Proposal based on Active Database Systems, Lic. Thesis, LiU-Tek-Lic 1996:54, Department of Computer and Information Science, Linköping University, Linköping, Sweden, 1996.

[45] Falkenroth E.: Computational Indexes for Time Series, in *the 8th International Conference on Scientific and Statistical Database Management*, Stockholm, Sweden, June 1996.

[46] Fielding R., Frystyk H., and Berners-Lee T.: Hypertext Transfer Protocol - HTTP/1.1, HTTP Working Group, INTERNET DRAFT, Nov. 22, 1995.

[47] Fishman D., 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, Risch T., Chan M. C., and Wilkinson W. K: Overview of the Iris DBMS, *Object-Oriented Concepts, Databases, and Applications*, ACM press, Addison-Wesley Publ. Comp., 1989.

[48] Flodin S. and Risch T.: Processing Object-Oriented Queries with Invertible Late Bound Functions. in *the 21st International VLDB Conference*, Zurich, Switzerland, Sept. 11-15, 1995.

[49] Forgy C. L.: Rete: A Fast Algorithm for the Many Pattern/ Many Object Pattern Match Problem, *Artificial Intelligence*, 19(1):17-34, 1982.

[50] Ghandeharizadeh S., Hull R., and Jacobs D.: Heraclitus : Elevating Deltas to be First-Class Citizens in a Database Programming Language, in *ACM Transactions on Database Systems*, Vol. 21, No. 3., September 1996, Pages 370-426.

[51] Garcia-Molina H. and Salem K.: Sagas, *ACM SIGMOD Conference*, San Francisco, California, 1987, Pages 249-259.

[52] Garcia-Molina H. and Salem K.: Main-Memory Database Systems: an Overview, *IEEE*

Transactions on Knowledge and Data Engineering, Vol. 4, No.6, Dec. 1992.

[53] Gatziu S. and Dittrich K. R: SAMOS: an Active Object-Oriented Database System, *IEEE Data Engineering Bulletin*, Vol. 15, No. 1-4, Dec. 1992, Pages 23-26.

[54] Gatziu S. and Dittrich K. R.: Events in an Active Object-Oriented Database System, in *Proceedings of the 1st International Workshop on Rules in Database Systems*, Edinbourgh, Scottland, 1993.

[55] Gehani N. H. and Jagadish H. V.: Ode as an Active Database: Constraints and Triggers, in *Proceedings of the 17th VLDB Conference.*, Barcelona, Spain, Sept. 1991, Pages 327-336.

[56] Gehani N. H., Jagadish H. V., and Shmueli O.: Composite Event Specification in Active Databases: Model & Implementation, in *Proceedings of the 18th VLDB Conference*, Vancouver, British Colombia, Canada, 1992, Pages 327-338.

[57] Georgakopoulos D., Hornick M., Sheth A.: An Overview of Workflow Management: From Process Modelling to Workflow Automation Infrastructure, *Distributed and Parallel Databases*, 3, 2, April 1995, Pages 119-153.

[58] Goebel V., Johansen B. H., Løchsen H. C., and Plagemann T.: Next Generation Database Technologies for Advanced Communication Services, in *Proceedings of the Third International Conference on Intelligence in Broadband Services and Networks - IS&N'95*, Herakliton, Crete, Greece, Oct. 1995, Pages 320-333.

[59] Gopal G. and Herman G.: Toward a Database-Driven Network, *IEEE* 1988.

[60] Grifeth N. D. and Lin Y-J: Extending Telecommunications Systems: The Feature-Interaction Problem, *Special Issue on Telecommunications, IEEE Computer Magazine*, Aug. 1993.

[61] Griffin T. and Libkin L.: Incremental Maintenance of Views with Duplicates, *ACM Sigmod Conference*, 1995, Pages 328-339.

[62] Gupta A. and Mumick I. S.: Maintenance of Materialized Views: Problems, Techniques and Applications, *IEEE Data Engineering bulletin*, Vol. 18, No. 2, 1995.

[63] Hanson E. N.: Rule Condition Testing and Action Execution in Ariel, *ACM SIGMOD Conference*, 1992, Pages 49-58.

[64] Hanson E. N. and Khosla S.: An Introduction to the TriggerMan Asynchronous Trigger Processor, in *Proceedings of the 3rd International Workshop on Rules in Database Systems - RIDS'97*, Skövde, Sweden, June 1997.

[65] Harrison J. V. and Dietrich S. W.: Condition Monitoring in an Active Deductive Database, Arizona State University, *ASU Technical Report* TR-91-022 (Revised), Dec. 1991.

[66] Hayes-Roth D., Washington R., Hewett R., Hewett M., and Seiver A.: Intelligent Monitoring and Control, *Proceedings of the 1989 International Joint Conference on Artificial Intelligence*, 1989.

[67] Hild S. G.: A Brief History of Mobile Telephony, University of Cambridge, Computer Laboratory, *Technical Report No. 372*, January 1995.

[68] Hedberg S. and Steizner M.: Knowledge Engineering Environment (KEE) System: Summary of Release 3.1, Intellicorp Inc. July 1987.

[69] Hurson A. R., Bright M. W., and Pakzad S. H.: Multidatabase Systems: An Advanced Solution for Global Information Sharing, *IEEE Computer Society Press*, ISBN-0-8186-4422-2, 1994.

[70] IEEE SCI Draft 2.00, SCI Scalable Coherent Interface, SCI: D2.00 P1596-18Nov91-doc233, 1991.

[71] Imielinski T. and Badrinath B. R.: Mobile Wireless Computing: Challanges in Data

Management, *Communications of the ACM*, 37(10), Oct. 1994.

[72] Imielinski T. and Korth H. F. (eds.): *Mobile Computing*, Kluwer Academic Publishers, ISBN: 0-7923-9697-9 , 1996.

[73] Ioannidis Y. E. and Cha Kang Y.: Randomized Algorithms for Optimizing Large Join Queries, *ACM SIGMOD Conference*, 1990, NJ, May 23-25, pp. 312-321.

[74] ITU (CCITT) X.722 Guidelines for the Management of Managed Objects (GDMO), Geneva 1992.

[75] Jensen C. S. and Mark L., Roussopoulos N.: Using Differential Techniques to Efficiently Support Transaction Time, in *the VLDB Journal*, Vol. 2, No. 1, Jan. 1993.

[76] Kalbfleisch C. W.: A MIB for Managing Web Servers, Internetwork, Sept. 1996, Page 28.

[77] Katiyar A. G. D. and Mumick I. S.: Maintaining Views Incrementally, *AT&T Bell Laboratories, Technical Report* 921214-19-TM, Dec. 1992.

[78] Keeler M.: Database of All Trades, *Database Programming & Design*, ISSN 0895-4518, Vol. 9, No. 11, Nov. 1996.

[79] Kersten M. L.: An Active Component for a Parallel Database Kernel, In *the Second International Workshop on Rules in Database Systems (RIDS'95)*, Athens, Greece, September 25-27, 1995, Springer Lecture Notes in Computer Science, ISBN 3-540-60365-4, Pages 277-291, 1995.

[80] Koenig S. and Paige R.: A Transformational Framework for the Automatic Control of Derived Data, In *Proceedings of VLDB Conference*, 1981, Pages 306-318.

[81] Lamy P.: M4 Interface Requirements and Logical MIB: ATM Network View, *The ATM Forum*, 1996.

[82] Litwin W. and Risch T.: Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates, *IEEE Transactions on Knowledge and Data Engineering* Vol. 4, No. 6, Dec. 1992.

[83] Loborg P. and Törne A.: A Hybrid Language for the Control of Multimachine Environments, in *the 4th Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert systems (IEA/AIE-91)*, (Hawaii, USA), June 1991.

[84] Lohman G. M., Lindsay B., Pirahesh H., and Schiefer K. B.: Extensions to Starburst: Objects, Types, Functions and Rules, *Communications of the ACM*, Oct. 1991, vol. 34, no. 10, Pages 94-109.

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

[86] Machani S-E.: Events in an Active Object-Relational Database System, Technical Report LiTH-IDA-Ex.9634, Department of Computer and Information Science, Linköping University, Sweden, 1996.

[87] Manna Z. and Pnueli A.: Models for Reactivity, *Acta Informatica*, 30:609-678, 1993.

[88] Mayer B.: *Eiffel - The Language*, Prentice-Hall, ISBN-0-13-247925-7, 1992.

[89] Medvinsky G. and Neuman B. C: . NetCash: A design for practical electronic currency on the Internet. In *Proceedings of 1st the ACM Conference on Computer and Communication Security*, Nov., 1993.

[90] Melton J.(ed.): ANSI SQL3 Papers SC21 N9463 - SC21 N9467, ANSI SC21 Secretariat, New York, USA, 1995.

[91] Miranker D. P.: TREAT: A Better Match Algorithm for AI Production Systems, *AAAI 87 Conference on Artificial Intelligence*, Aug. 1987, Pages 42-47.

[92] Morgenstern M.: Active Databases as a Paradigm for Enhanced Computing Environ-

ments, *Proceedings of the 9th VLDB Conference*, Florence, Nov. 1983.

[93] Ohsie D., Dewan M. D., Stolfo S. J., and Da Silva S.: Performance of Incremental Update in Database Rule Processing, *RIDE ADS'94*, Houston, Februari, 1994.

[94] Özsu M. T. and Valduriez P.: *Principles of Distributed Database Systems*, Prentice Hall Inc., ISBN 0-13-715681-2, 1991.

[95] Padron-McCarthy T. and Risch T.: Performance-Polymorphic Execution of Real-Time Queries. In *the First Workshop on Real-Time Databases: Issues and Applications (RTDB-96)*, Newport Beach, CA, March 7-8, 1996.

[96] Paige R. and Koenig S.: Finite Differencing of Computable Expressions, In *ACM Transactions on Programming Languages and Systems*, 4(2):402-454, 1992.

[97] Paton N. W., Doan K., Díaz O., and Jaime A.: Exploitation of Object-Oriented and Active Constructs in Database Interface Development, *IDS 1996*: 1.

[98] Prasad Sistla A. and Wolfson O.,: Temporal Triggers in Active Databases, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7., No. 3, June 1995.

[99] Qian X. and Wiederhold G.: Incremental Recomputation of Active Relational Expressions, *IEEE Transactions on Knowledge and Data Engineering* Vol. 3, No. 3 , Dec. 1991, Pages 337-341.

[100] Quass D.: Maintenance Expressions for Views with Aggregation, in *Workshop on Materialized Views: Techniques and Applications*, in cooperation with ACM SIGMOD, Montreal, Canada, June 1996.

[101] Ramamritham K.: Real-Time Databases, *Distributed and Parallel Databases*, Kluwer Academic Publishers, Vol. 1, 1993, Pages 199-226.

[102] Risch T.: Monitoring Database Objects, In *the Proceedings of 15th VLDB Conference*, Amsterdam, 1989.

[103] Rosenthal A., Chakravarthy S., Blaustein B., and Blakely J.: Situation Monitoring for Active Databases, In *Proceedings of the 15th VLDB Conference*, Amsterdam, 1989, Pages 455-464.

[104] Sandewall E.: *Features and Fluents. The Representation of Knowledge about Dynamical Systems. Volume I.* Oxford Univ. Press, 1994.

[105] Saran A., Sastri A., Agrawal D., El Abbadi A., and Smith T.R.: Experiences in the Design of a Kernel for Computational and Modelling Systems, in *the 6th International Conference on the Management of Data*, India, Dec. 1994.

[106] Schwiderski S.: Monitoring the Behaviour of Distributed Systems, Ph. D Thesis, Selwyn College, University of Cambridge, Cambridge 1996.

[107] Selinger P., Astrahan M. M., Chamberlin R.A., Lorie R. A., and Price T.G.: Access Path Selection in a Relational Database Management System, *ACM SIGMOD Conference*, Boston, MA, June 1979, Pages 23-54.

[108] Shipman D. W.: The Functional Data Model and the Data Language Daplex, *ACM Transactions on Database Systems*, 6(1), 3, 1981.

[109] Shoshani A., Olken F., and Wong H. K. T., Characteristics of Scientific Databases, *Proceedings of the 10th VLDB Conference*, Singapore, Aug. 1984.

[110] Sköld M. and Risch T.: Using Partial Differencing for Efficient Monitoring of Deferred Complex Rule Conditions, in *Proceedings of the 12th International Conference on Data Engineering (ICDE'96)*, New Orleans, Louisiana, Feb. 1996.

[111] Snodgrass R. and Ahn I.: Temporal Databases, in *IEEE Computer*, Pages 35-42, Sept. 1986.

[112] Spilker J. and Parkinson B. (eds.) : *Global Positioning System: Theory and Applica-*

tions, AIAA, 1997.

[113] Stacey D.: Replication: DB2, Oracle, or Sybase, *SIGMOD Record* Vol. 24, No. 4, Dec. 1995, Pages 95-98.

[114] Stallings W.: SNMP, SNMPv2 and CMIP: *The Practical Guide to Network Management Standards*, Addison-Wesley Publishing Co. Inc., ISBN 0-201-63331-0, 1993.

[115] Stevens W.R.: TCP/IP Illustrated, Volume 1, Addison Wesley, ISBN 0-201-63346-9, Oct. 1995.

[116] Stonebraker M. and Row L.: The Design of POSTGRES, In *Proceedings of ACM SIGMOD Conference*, Washington, D.C., May 1986, Pages 340-355.

[117] Stonebraker M., Jhingran A., Goh J., and Potamianos S.: On Rules, Procedures, Caching, and Views in Database Systems, In *Proceedings of ACM SIGMOD Conference*, 1990, Pages 281-290.

[118] Stonebraker M. and Moore D.: *Object-Relational DBMSs - The Next Great Wave*, Morgan Kaufmann Publishers, Inc., ISBN 1-55860-397-2, 1996.

[119] Su S. Y. W., Lam H., Yu T-F., Arroyo-Figueroa J. A., Yang Z., and Lee S.: NCL - A Common Language for Achieving Rule-Based Interoperabillity Among Heterogeneous Systems, *Journal of Intelligent Information Systems*, Vol. 6, No. 2/3, May 1996.

[120] Taivalsaari A.: On the Notion of Inheritance, *ACM Computing Surveys*, Vol. 28, No. 3 Sept. 1996, Pages 438-479.

[121] Tanenbaum A. S.: *Computer Networks, Second Edition*, Prentice-Hall International Inc., 1989, ISBN 0-13-166836-6.

[122] Torbjørnsen Ø.: Multi-Site Declustering Strategies for Very High Database Service Availability, Ph. D. Thesis, Doktor Ingeniøravhandling 1995:16, Dept. of Computer Systems and Telematics, Norwegian Institute of Technology, University of Trondheim, Norway 1995.

[123] Törne A.: The Instruction and Control of Multi-Machine Environments, in *Proceedings of the 5th International Conference on Applications of Artificial Intelligence in Engineering*, Vol. 2, Springer Verlag, 1990.

[124] Tving I.: Multiprocessor Interconnection using SCI, Master Thesis, DTH ID-E 579, Dept. of Comp. Science, the Technical University of Denmark, Denmark 1992.

[125] Ullman J. D.: *Principles of Database and Knowledge-Base Systems, Volume I & II*, Computer Science Press, 1988, 1989.

[126] Urpí T. and Olivé A.: A Method for Change Computation in Deductive Databases, in *Proceedings of the 18th International VLDB Conference*, Vancouver, 1992, Pages 225-237.

[127] Varshney U.: Supporting Mobility with Wireless ATM, *IEEE Computer*, Jan. 1997, Pages 131-133.

[128] Venkatrao M. and Pizzo M.: SQL/CLI- A New Binding Style For SQL, *SIGMOD Record* Vol. 24, No. 4, Dec. 1995, Pages 72-77.

[129] Wang Y-W. and Hanson E. N.: A Performance Comparison of the Rete and TREAT Algorithms for Testing Database Rule Conditions, In *Proceedings of the International Conference on Data Engineering (ICDE) 1992*, Arizona, Feb. 1992, Pages 88-97.

[130] Werner M.: Multidatabase Integration using Polymorphic Queries and Views, Licentiate Thesis No 546, Department of Computer and Information Science, Linköping University, Sweden, 1996.

[131] Wiederhold G.: Mediators in the Architecture of Future Information Systems, *IEEE Computer*, March 1992.

[132] Widom J. and Finkelstein S.J.: Set-oriented production rules in relational database system, *ACM SIGMOD Conference*, Atlantic City, New Jersey 1990, Pages 259-270.

[133] Widom J. and Ceri S. (ed.): *Active Database Systems - Triggers and Rules for Advanced Database Processing*, Morgan Kaufmann Publishers, Inc., ISBN-1-55860-304-2, 1996.

[134] Wolski A., Karvonen J., and Puolakka A.: The RAPID Case Study: Requirements for and the Design of a Fast-Response Database System, in *Proceedings of the First International Workshop on Real-Time Databases: Issues and Applications*, Newport Beach, California, USA, March 1996, Pages 32-39.

[135] Yang Z., Duddy K.: CORBA: A Platform for Distributed Object Computing, *ACM Operating Systems Review*, Volume 30, No. 2, April 1996.

[136] Zbigniew M. (ed.): *Statistical and Scientific Databases*, The Ellis Horwood Limited, ISBN 0-13-850652-3, 1991.

[137] Zhuge Y., Wiener J. L., and Garcia-Molina H.: Multiple View Consistency for Data Warehousing, in *Proceedings of the 13th International Conference on Data Engineering (ICDE)*, Birmingham, UK, 1997, Pages 289-300.

14 Appendix

14.1 The Current Rule Syntax in AMOS

The final syntax (when this thesis was printed) of the rules in AMOS are as follows:

14.1.1 Rule Creation and Deletion

create rule *rule-name parameter-specification*
[*for-each-clause*]
[**on** *event-type-specification*]
[**when** *predicate-expression*]
do *procedure-expression*

delete rule *rule-name*

where

for-each-clause ::=
for each *variable-declaration-commalist*

event-type-specification ::=
added(*function-call*) |
removed(*function-call*) |
updated(*function-call*) |
created(*variable-name*) |
deleted(*variable-name*)[1] |
foreign-event-name |
event-type-specification **and** *event-type-specification* |
event-type-specification **or** *event-type-specification* |
event-type-specification **before** *event-type-specification* |
event-type-specification **after** *event-type-specification*

14.1.2 Rule Activation and Dectivation

activate rule *rule-name* ([*parameter-value-commalist*])

1. Actually not yet implemented due to technical problems in AMOS on how to reference objects that have been marked as deleted.

[**strict**]
[**priority** 0|1|2|3|4|5]
[**into** *context-name*]

deactivate rule *rule-name* ([*parameter-value-commalist*])
[**from** *context-name*]

14.1.3 Rule Checking

check();

The rules in the default deferred context will be checked.

check(:context);

The rules activated into the specified context will be checked.

commit;

The deferred context will be checked, then the transaction will be committed, and finally the detached context will be checked.

14.1.4 Rule Contexts and Sagas

create context *context-name*

delete context *context-name*

activate context *context-name*

deactivate context *context-name*

create saga *saga-name*

create subsaga *saga-name*

A subsaga will be created (within the current saga) that will cause nesting of compensations.

delete saga *saga-name*

saga *saga-name procedure-body*
compensation *procedure-body*

The first procedure body will be executed and committed as a separate transaction and the second procedure body will be registered as the corresponding compensating transaction.

commit_saga(:saga);

abort_saga(:saga);

The specified compensations will be executed in reverse order.

stop_compensation();

The ongoing chain of compensations will be aborted.

associate(:context, :saga, check_mode);

The associated contexts will be automatically activated when the specified saga is entered, checked when transactions of the saga are committed (if check_mode is “exit”), and deactivated when the saga is exited. When a saga is entered again, the associated contexts are reactivated. Alternatively, the attached contexts can be checked when the complete saga is committed (if check_mode is “commit”). When a saga is aborted, associated rule contexts can also be checked during the compensating transactions (if check_mode is “roll-back”).

14.1.5 Creation Foreign Data Sources

Foreign events are supported in AMOS ECA-rules. Currently the foreign events can be specified by hand by registering a new event type by name with the event manager along with an event function that stores the event data. The new event can be manually raised by:

raise(<event-name>, <event-data>);

Raises the event and associates the transaction time and the specified event-data with the event[1].

A future extension of this will be to automatically create foreign events as part of the foreign function definitions. This is not yet implemented, but could look something like:

create function *function-name parameter-specification result-specification*
as foreign *foreign-data-source-access-specification*
[**set** *foreign-data-source-change-specification*]
[**add** *foreign-data-source-change-specification*]
[**remove** *foreign-data-source-change-specification*]
[**event** *foreign-event-name*]

where

foreign-data-source-access-specification ::=
[*implementation-specification*]
[**size** *integer*]
[**push** *interrupt-method* [*file-descriptor* | *address*]]
[**pull frequency** *time*]
[**transformations** *transformation-specification*]
[**costhint** *optimization-specification*]

foreign-data-source-change-specification ::=
function-call

1. The raise operation is also available through the AMOS fast path application interface.

[**event** *foreign-event-name*]

14.2 The Relational Operators in Datalog

Datalog, or *domain calculus*, is equivalent to relational calculus in expressional power. The relational operations *union*, *difference*, *cartesian product*, *selection* and *projection* can be directly specified in Datalog. Other operations such as *join* and *intersection* that can be derived from these basic operations can also be directly specified.

Union

PARENT = FATHER $\cup$ MOTHER

is translated into

`parent(X, Y)` $\leftarrow$ `father(X, Y)` $\vee$ `mother(X, Y)`

or

`parent(X, Y)` $\leftarrow$ `father(X, Y)`

`parent(X, Y)` $\leftarrow$ `mother(X, Y)`

Difference

FATHER = PARENT - MOTHER

is translated into

`father(X, Y)` $\leftarrow$ `parent(X, Y)` $\wedge \neg$`mother(X, Y)`

Cartesian product

PAIR = PERSON $\times$ PERSON

is translated into

`pair(X, Y)` $\leftarrow$ `person(X)` $\wedge$ `person(Y)`

Selection

PAIR = $\sigma_{\$1 \neq \$2}$(PERSON$_{\$1}$ $\times$ PERSON$_{\$2}$)

MILLIONAIRE = $\sigma_{\$2 > 999999}$INCOME$_{\$1,\$2}$

is translated into

`pair(X, Y)` $\leftarrow$ `person(X)` $\wedge$ `person(Y)` $\wedge$ `X` $\neq$ `Y`

`millionaire(X)` $\leftarrow$ `income(X, Y)` $\wedge$ `Y` $>$ `999999`

Projection

$IS\_FATHER = \pi_{\$1}FATHER_{\$1}$

is translated into

```
is_father(X) ← father(X, Y)
```

Join

$GRANDPARENT = \pi_{X,Z}PARENT_{X,Y} \bowtie PARENT_{Y,Z} =$

$\pi_{X,Z}(\sigma_{Y1=Y2}PARENT_{X,Y1} \times PARENT_{Y2,Z})$

is directly translated into

```
grandparent(X, Z) ←
            parent(X, Y1) ∧ parent(Y2, Z) ∧ Y1 = Y2
```

or more naturally expressed as

```
grandparent(X, Z) ← parent(X, Y) ∧ parent(Y, Z)
```

Intersection

$RICH\_GRANDPARENT = GRANDPARENT \cap MILLIONAIRE =$

$(GRANDPARENT \cup MILLIONAIRE) -$

$((GRANDPARENT - MILLIONAIRE) \cup$

$(MILLIONAIRE - GRANDPARENT))$

is directly translated into

```
rich_grandparent(X) ←
            grandparent(X) ∨ millionaire(X) ∧
            ¬((grandparent(X) ∧ ¬millionaire(X)) ∨
              (millionaire(X) ∧ ¬grandparent(X))
```

or more naturally expressed as

```
rich_grandparent(X) ← grandparent(X) ∧ millionaire(X)
```

14.3 Justification for Partial Differencing

Below follows a formal justification for the correctness of partial differencing.

There exists an isomorphism f, denoted $\cong_f$, between the boolean algebra of ObjectLog and set algebra [1]:

$f: <O, \neg, \wedge, \vee> \rightarrow <2^{At(O)}, \sim, \bigcap, \bigcup>,$

where O is the domain of objects in the database, $\neg$ is negation based on the Closed World Assumption, $\wedge$ is logical conjunction, $\vee$ is logical disjunction, $2^{At(O)}$ is the power set of atoms in O, $\sim$ is set complement, $\bigcap$ is set intersection, and $\bigcup$ is set

union. Using this we can define change monitoring of ObjectLog through set operations.

Let Δ_+S, delta-plus of S, be the set of additions (positive changes) to a set S and Δ_-S, delta-minus of S, the set of deletions (negative changes) from S. Let the Δ-*set* (delta-set) of S be a tuple of the positive and the negative changes of a set S:

$$\Delta S = <\Delta_+S, \Delta_-S>$$

Let $\bigcup_\Delta$ (delta-union) be the operator that calculates the union of two Δ-sets:

$$\Delta P_1 \bigcup_\Delta \Delta P_2 = <(\Delta_+P_1 \cup \Delta_+P_2) - (\Delta_-P_1 \cup \Delta_-P_2), \\ (\Delta_-P_1 \cup \Delta_-P_2) - (\Delta_+P_1 \cup \Delta_+P_2) >$$

To detect changes of derived relations we define conjunction, disjunction, and negation in terms of their differentials as:

$$\Delta(Q \wedge R) \cong_f \Delta(Q \cap R) = \\ <(\Delta_+Q \cap R) \cup (Q \cap \Delta_+R), \{\}> \\ \bigcup_\Delta \\ <\{\}, (\Delta_-Q \cap R_{old}) \cup (Q_{old} \cap \Delta_-R>$$

or for bag-oriented semantics

$$\Delta(Q \wedge R) \cong_f \Delta(Q \cap R) = \\ <(\Delta_+Q \cap R) \cup ((Q - \Delta_+Q) \cap \Delta_+R), \{\}> \\ \bigcup_\Delta \\ <\{\}, (\Delta_-Q \cap R_{old}) \cup ((Q_{old} - \Delta_-Q) \cap \Delta_-R)>$$

$$\Delta(Q \vee R) \cong_f \Delta(Q \cup R) = \\ <(\Delta_+Q - R_{old}) \cup (\Delta_+R - Q_{old}), \{\}> \\ \bigcup_\Delta \\ <\{\}, (\Delta_-Q - R) \cup (\Delta_-R - Q)>$$

$$\Delta(\neg Q) \cong_f \Delta(\sim Q) = <\Delta_-Q, \Delta_+Q>$$

where $R_{old} = (\Delta_-R \cup R) - \Delta_+R$
and since $\Delta_+R \cap \Delta_-R = \emptyset$, i.e. $\Delta_-R - \Delta_+R = \Delta_-R$, we have
$R_{old} = \Delta_-R \cup (R - \Delta_+R) = \Delta_-R \cup (R \cap \sim\Delta_+R)$ which can be expressed logically by:

$$R_{old} = \Delta_-R \vee (R \wedge \neg(\Delta_+R))$$

where Q_{old} is defined likewise.

Let D_p be the set of all relations that a relation P depends on. Let the *positive partial differentials* $\Delta P/\Delta_+X$ of a relation P be defined by the body of P where a single relation $X \in D_p$ has been substituted by its positive Δ-relation Δ_+X.

Let the *negative partial differentials* $\Delta P/\Delta_-X$ of a relation P be defined by the body of P where a single relation $X \in D_p$ has been substituted by its negative Δ-relation Δ_-X and where all $Y \in D_p$, $Y \neq X$, have been substituted by Y_{old}.

Positive partial changes are combined by:
$\Delta_+P = \bigcup \Delta P/\Delta_+X, \forall X \in D_p$
and negative changes by
$\Delta\_P = \bigcup \Delta P/\Delta\_X, \forall X \in D_p$
The full *differential* (delta-relation) is defined as:
$\Delta P = <\Delta_+P, \{\}> \bigcup_\Delta <\{\}, \Delta\_P>$

Correctness is here defined as: given a relation P where D_p is the set of all other relations that P depends on and that we have all the net changes ΔS of all relations $S \in D_p$, then ΔP reflects the changes to P.

1. If P is a base relation then its changes can be found directly in ΔP.

2. If P is a derived, conjunctive relation then:
   i) If $P \leftarrow S \wedge T$, then we need to show that $\Delta P/\Delta_+S \leftarrow \Delta_+S \wedge T$ for all positive changes to S.
   If T is a base relation, then since the contribution of deduced facts in P are dependent on the facts both in S and T, then any added facts in S that are also in T are also in P. In some cases and when using set-oriented semantics, added facts in S might give deduced facts that were already present in P_{old}, then the algorithm might cause *nervous* triggering of rules. To avoid this we have to calculate $\Delta P/\Delta_+S - P_{old}$. If T is a derived relation of n conjunctions then clearly:
   $\Delta P/\Delta_+S \leftarrow \Delta_+S \wedge T_1 \wedge ... \wedge T_n$
   If T is a derived relation of n+1 conjunctions, then we also have:
   $\Delta P/\Delta_+S \leftarrow \Delta_+S \wedge T_1 \wedge ... \wedge T_{n+1}$
   and by induction the execution of positive, conjunctive partial Δ-relations has been shown to be correct.

   ii) If $P \leftarrow S \wedge T$, then we need to show that $\Delta P/\Delta\_S \leftarrow \Delta\_S \wedge T_{old}$ for all negative changes to S.
   If T is a base relation, then since the contribution of deduced facts in P are dependent on the facts both in S and T, then any removed facts from S that also where in T_{old} supported facts are facts that are no longer in P. In some cases and when using set-oriented semantics, removed facts from S might give deduced facts that are still present in P. To avoid incorrect propagation of negative changes we have to check that the deduced change is not still present in P, i.e. $\Delta P/\Delta\_S - P$.
   If T is a derived relation of n conjunctions, then clearly:
   $\Delta P/\Delta\_S \leftarrow \Delta\_S \wedge T_{old\ 1} \wedge ... \wedge T_{old\ n}$
   If T is a derived relation of n+1 conjunctions, then we also have:
   $\Delta P/\Delta\_S \leftarrow \Delta\_S \wedge T_{old\ 1} \wedge ... \wedge T_{old\ n+1}$
   and by induction the execution of negative, conjunctive partial differentials has been shown to be correct.

3. If P is a derived, disjunctive relation, in disjunctive normal form, (and assuming set-oriented semantics), then:

i) If $P \leftarrow S \vee T$, then we need to show that $\Delta P/\Delta_+ S \leftarrow \Delta_+ S \wedge \neg T_{old}$ for all positive changes to S.
If T is a base relation, then since the contribution of deduced facts in P are dependent on facts in S or T, then any added facts to S will cause positive changes to P if T was not already true for those facts.
If T is a derived relation of n disjuncts, then clearly:
$\Delta P/\Delta_+ S \leftarrow \Delta_+ S \wedge \neg T_{old\ 1} \wedge ... \wedge \neg T_{old\ n}$
If T is a derived relation of n+1 disjuncts, then we also have:
$\Delta P/\Delta_+ S \leftarrow \Delta_+ S \wedge \neg T_{old\ 1} \wedge ... \wedge \neg T_{old\ n+1}$
and by induction the execution of positive, disjunctive partial differentials has been shown to be correct.

ii) If $P \leftarrow S \vee T$, then we need to show that $\Delta P/\Delta\_ S \leftarrow \Delta\_ S \wedge \neg T$ for all negative changes to S.
If T is a base relation, then since the contribution of deduced facts in P are dependent on facts in S or T, then any removed facts from S will cause negative changes to P if T is not true for those facts.
If T is a derived relation of n disjuncts then clearly:
$\Delta P/\Delta\_ S \leftarrow \Delta\_ S \wedge \neg T_1 \wedge ... \wedge \neg T_n$
If T is a derived relation of n+1 disjuncts then we also have:
$\Delta P/\Delta\_ S \leftarrow \Delta\_ S \wedge \neg T_1 \wedge ... \wedge \neg T_{n+1}$
and by induction the execution of negative, disjunctive partial differentials has been shown to be correct.

4. If P is a derived negated relation, $P \leftarrow \neg S$, then we need to show that:

i) $\Delta P/\Delta\_ S \leftarrow \Delta_+ S$
All facts not in S are deduced to be in P. If a fact is added to S, then a negative change has to be deduced for P.

ii) $\Delta P/\Delta_+ S \leftarrow \Delta\_ S$
All facts in S are deduced to not be in P. If a fact is removed from S, then a positive change has to be deduced for P.

5. If P is a derived relation that depends on the subrelations D_p, then the changes calculated by $\Delta P/\Delta_+ X$ and $\Delta P/\Delta\_ X$, $X \in D_p$, can be combined by $\cup_\Delta$ to give the total changes of P.

i) For set-oriented semantics $\cup_\Delta$ is defined as joining positive and negative changes in Δ-sets by removing duplicates and extinguishing complementary positive and negative changes.

ii) For bag-oriented semantics $\cup_\Delta$ is defined as joining positive and negative sets by keeping a count of duplicates and extinguishing complementary positive and negative changes. For conjunctions a modification of partial differentials will also have to be done to remove overlaps in the execution.
Positive changes are calculated by:
changing all subgoals Y in $\Delta P/\Delta_+ X$ to $Y - \Delta_+ Y$, $\forall X, Y \in D_p$ and $X \neq Y$ and

where Y precedes Δ_+X in the conjunction,
and negative changes by:
changing all Y_{old} in $\Delta P/\Delta\_X$ to Y_{old} - Δ_Y, $\forall X, Y \in D_p$ and $X \neq Y$ and where Y_{old} precedes Δ_X in the conjunction.

In the proof above an assumption was made that we have the net changes of the relation S collected in ΔS. The collection of changes of a relation was defined using the $\cup_\Delta$ operator. If relations are defined to have set-oriented semantics, then the order of accumulation of changes has to be the same as the changes occurred in the transaction.

The proof above can be used for calculating incremental changes to the relational operators (with the related parts of the proof in parenthesis):

Union: (1, 3, 5)

$\texttt{parent(X, Y)} \leftarrow \texttt{father(X, Y)} \vee \texttt{mother(X, Y)}$

$\Delta\texttt{parent(X, Y)}/\Delta_+\texttt{father} \leftarrow \Delta_+\texttt{father(X, Y)} \wedge \neg\texttt{mother}_{old}\texttt{(X, Y)}$

$\Delta\texttt{parent(X, Y)}/\Delta_+\texttt{mother} \leftarrow \neg\texttt{father}_{old}\texttt{(X, Y)} \wedge \Delta_+\texttt{mother(X, Y)}$

$\Delta\texttt{parent(X, Y)}/\Delta\texttt{\_father} \leftarrow \Delta\texttt{\_father(X, Y)} \wedge \neg\texttt{mother(X, Y)}$

$\Delta\texttt{parent(X, Y)}/\Delta\texttt{\_mother} \leftarrow \neg\texttt{father(X, Y)} \wedge \Delta\texttt{\_mother(X, Y)}$

Difference: (1, 2, 4, 5)

$\texttt{father(X, Y)} \leftarrow \texttt{parent(X, Y)} \wedge \neg\texttt{mother(X, Y)}$

$\Delta\texttt{father(X, Y)}/\Delta_+\texttt{parent} \leftarrow \Delta_+\texttt{parent(X, Y)} \wedge \neg\texttt{mother(X, Y)}$

$\Delta\texttt{father(X, Y)}/\Delta_+\texttt{mother} \leftarrow \texttt{parent(X, Y)} \wedge \Delta\texttt{\_mother(X, Y)}$

$\Delta\texttt{father(X, Y)}/\Delta\texttt{\_parent} \leftarrow \Delta\texttt{\_parent(X, Y)} \wedge \neg\texttt{mother}_{old}\texttt{(X, Y)}$

$\Delta\texttt{father(X, Y)}/\Delta\texttt{\_mother} \leftarrow \texttt{parent}_{old}\texttt{(X, Y)} \wedge \Delta_+\texttt{mother(X, Y)}$

Cartesian product: (1, 2, 5)

$\texttt{pair(X, Y)} \leftarrow \texttt{person(X)} \wedge \texttt{person(Y)}$

$\Delta\texttt{pair(X, Y)}/\Delta_+\texttt{person'} \leftarrow \Delta_+\texttt{person(X)} \wedge \texttt{person(Y)}$

$\Delta\texttt{pair(X, Y)}/\Delta_+\texttt{person''} \leftarrow \texttt{person(X)} \wedge \Delta_+\texttt{person(Y)}$

$\Delta\texttt{pair(X, Y)}/\Delta\texttt{\_person'} \leftarrow \Delta\texttt{\_person(X)} \wedge \texttt{person}_{old}\texttt{(Y)}$

$\Delta\texttt{pair(X, Y)}/\Delta\texttt{\_person''} \leftarrow \texttt{person}_{old}\texttt{(X)} \wedge \Delta\texttt{\_person(Y)}$

Selection: (1, 2, 5)

$\texttt{millionaire(X)} \leftarrow \texttt{income(X, Y)} \wedge \texttt{Y} > \texttt{999999}$

$\Delta\texttt{millionaire(X)}/\Delta_+\texttt{income} \leftarrow \Delta_+\texttt{income(X, Y)} \wedge \texttt{Y} > \texttt{999999}$

$\Delta\texttt{millionaire(X)}/\Delta\texttt{\_income} \leftarrow \Delta\texttt{\_income(X, Y)} \wedge \texttt{Y} > \texttt{999999}$

Projection: (1, 5)

$\texttt{is\_father(X)} \leftarrow \texttt{father(X, Y)}$

$\Delta\texttt{is\_father(X)}/\Delta_{+}\texttt{father} \leftarrow \Delta_{+}\texttt{father(X, Y)}$

$\Delta\texttt{is\_father(X)}/\Delta_{-}\texttt{father} \leftarrow \Delta_{-}\texttt{father(X, Y)}$

Join: (1, 2, 5)

$\texttt{grandparent(X, Z)} \leftarrow \texttt{parent(X, Y)} \wedge \texttt{parent(Y, Z)}$

$\Delta\texttt{grandparent(X, Z)}/\Delta_{+}\texttt{parent'} \leftarrow$
$\texttt{.} \qquad \Delta_{+}\texttt{parent(X, Y)} \wedge \texttt{parent(Y, Z)}$

$\Delta\texttt{grandparent(X, Z)}/\Delta_{+}\texttt{parent''} \leftarrow$
$\texttt{parent(X, Y)} \wedge \Delta_{+}\texttt{parent(Y, Z)}$

$\Delta\texttt{grandparent(X, Z)}/\Delta_{-}\texttt{parent'} \leftarrow$
$\Delta_{-}\texttt{parent(X, Y)} \wedge \texttt{parent}_{\text{old}}\texttt{(Y, Z)}$

$\Delta\texttt{grandparent(X, Z)}/\Delta_{-}\texttt{parent''} \leftarrow$
$\texttt{parent}_{\text{old}}\texttt{(X, Y)} \wedge \Delta_{-}\texttt{parent(Y, Z)}$

Intersection: (1, 2, 5)

$\texttt{rich\_grandparent(X)} \leftarrow \texttt{grandparent(X)} \wedge \texttt{millionaire(X)}$

$\Delta\texttt{rich\_grandparent(X)}/\Delta_{+}\texttt{grandparent} \leftarrow$
$\Delta_{+}\texttt{grandparent(X)} \wedge \texttt{millionaire(X)}$

$\Delta\texttt{rich\_grandparent(X)}/\Delta_{+}\texttt{millionaire} \leftarrow$
$\texttt{grandparent(X)} \wedge \Delta_{+}\texttt{millionaire(X)}$

$\Delta\texttt{rich\_grandparent(X)}/\Delta_{-}\texttt{grandparent} \leftarrow$
$\Delta_{-}\texttt{grandparent(X)} \wedge \texttt{millionaire}_{\text{old}}\texttt{(X)}$

$\Delta\texttt{rich\_grandparent(X)}/\Delta_{-}\texttt{millionaire} \leftarrow$
$\texttt{grandparent}_{\text{old}}\texttt{(X)} \wedge \Delta_{-}\texttt{millionaire(X)}$

15 The Papers

15.1 Paper I

P. Loborg, P. Holmbom, M. Sköld, and A. Törne: A Model for the Execution of Task Level Specifications for Intelligent and Flexible Manufacturing Systems, in Proceedings of the Vth International Symposium on Artificial Intelligence, ISAI92,Cancun, Mexico, Dec. 7-11, 1992. Also published in Journal of Integrated Computer-Aided Engineering (special issue on AI in Manufacturing and Robotics).

A MODEL FOR THE EXECUTION OF TASK-LEVEL SPECIFICATIONS FOR INTELLIGENT AND FLEXIBLE MANUFACTURING SYSTEMS

Peter Loborg, Per Holmbom*, Martin Sköld and Anders Törne

Department of Computer and Information Science
*Department of Physics and Measurement Technology
Linköping University, Sweden

ABSTRACT

We introduce here a software architecture for the control of a sensor-based manufacturing system consisting of a number of machines and peripheral equipment. The architecture divides the programming effort into two levels, task-level programming and control-level programming. The task-level programming is based on the programming of a discrete model of the world, the World Model (WM). The WM provides a symbolic representation of the world state and isolates the task programs from the control level algorithms. Programming the control level amounts to modelling the manufacturing equipment as components with 'behaviour' using object oriented techniques. Each component specifies how it should react to changes in the WM, i.e. selection and specification of the control algorithms to be executed. Programming at both levels can be done incrementally and control algorithms may be changed dynamically in the real-time kernel.

INTRODUCTION

This paper presents a model for the execution of task level specifications in a manufacturing system. The work presented has been carried out within the ARAMIS-project[1] and is a part of a cooperation project between the Dept. of Computer Science (IDA) and the Dept. of Physics and Measurement Technology (IFM) at Linköping University.

The motivation behind this project has been to provide control engineers with adequate tools to design and specify intelligent behaviours, i.e. behaviours which from the engineer's point of view are robust to changes in the environment and perform a required high-level function without additional detailed low level control programming. By control engineers we mean any engineer or operator who is specifying or diagnosing the behaviour of a physical system, with one or several embedded computers/control systems. More specifically,

1. The ARAMIS-project at CAELAB, Department of Computer Science, Linköping University, Sweden, is the continuation of a joint research project between the Department of Computer Science, ABB Corporate Research and ABB Robotics in Västerås, Sweden, during the years 1985 -1988.

we focus on manufacturing systems consisting of a number of machines and peripheral equipment. This kind of environment is characterized by high demands on the precision and accuracy of movements, complicated and interdependent sequential actions and different user categories performing different tasks with respect to the equipment. The users must therefore be provided with means to specify accurate control algorithms as well as task level programs. These programming tasks are also normally performed by different user categories.

The function of an embedded control system is to provide the engineer with the means to execute actions in the environment - automatically by preprogramming or manually by interactive command. We are using 'programming' to denote not just the actual programming of the computer, but also the specification of work descriptions and operation lists. The engineer specifies in the program a "sequence" of actions to perform. This program is very often cyclic, e.g. a feedback algorithm for fast motoric actions or a manufacturing cycle in a machine cell. The former is an example of an action which traditionally is specified at a low level, close to the controlled process and involving explicit reference to I/O on the embedded computer. The latter, on the other hand, is an example of a program which is originally specified at a higher level involving basic operations of larger time granularity. More importantly, the latter type of program is specified without reference to the embedded control systems, e.g. as operations lists or work descriptions. The translation of such specifications to executable programs in the embedded computers is today performed manually as a specific programming task.

Our aim is to equip the embedded computer system with the ability to execute "task programs", like operations lists and work descriptions, directly, i.e. behaving intelligently. The different user categories and abilities, the different primitive elements in the specification languages and the different types of abstract machines for task programs and control algorithms imply some sort of multi-level programming and executing environment. For the moment we are investigating the possibilities of just three levels, which should provide enough evidence for the usability of the ideas. The levels are - the task level, the control level and the physical level. Three levels might also be a just compromise between functionality and complexity in the programming and run-time environments.

First we present a general view on sensing and sensor fusion which motivates the concept of sensor integration and the ARAMIS-model of the environment and task program execution. Next the ARAMIS-model is described in more detail. After this, a presentation is made about the specification and encapsulation of control algorithms, together with an example. Finally we discuss the current status of the project and some future work.

We will not discuss the task-level language and the higher level functionality in this paper. The general ideas regarding the task level are presented in [14] and the graphical task level language for specifying intelligent behaviours is presented in [7].

SENSOR INTEGRATION AND PROGRAMMING IN PHYSICAL ENVIRONMENTS

Sensor fusion is characterized by Clark and Yuille [2] as concerned with methods for combining raw sensory data to obtain information about the world. They classify the methods into weakly and strongly coupled, where the weakly coupled assume independence of data sources. Dependence means that the algorithm and its validity constraints for calculating one world (fused sensor-) data is dependent on the output of another, separately calculated, world data, e.g. feature matching stereo algorithms which are dependent on the depth information calculated from other sources.

Our concept of sensor integration extends this dependence to also comprise the dependence of algorithms on implicit knowledge about the world state and not only on measurable, sensed knowledge. This implicit knowledge is derived from the knowledge of the executing context of the actions performed in the environment.

Example 1:

Consider a program which can switch the light ON and OFF in the room. No sensory device exists for light detection and the operation of the switch is failproof (within the validity of the system specification). If there is a position determination task in the program, it might have different fusion algorithms for darkness and light. This information is implicit in the sense that it is determined by the last switching command and is non-local to the position sensing device. The program using the ON/OFF information is independent of whether this information is explicitly measured or implicitly calculated from execution context information.

We stress the point that the programming environment should support a unified view on explicitly sensed and implicitly known information in the programming of the (intelligent) behaviour of the complete system. To clarify this point, consider the example again. An operator or an engineer who is interested in using the position function in the description of the behaviour would like the program to be independent of whether there exists a sensor for light or of the type of sensor used. This abstraction barrier is realized by the concept of world model programming explained below. We are convinced that such abstraction barriers are desirable in industrially used control equipment, where several different user categories cooperate in supervision and programming.

Our model has similarities with current implementations of different integrated environments for programming (“controlling”) computers where the machine is an abstract computing device designed for the portability of programs. However, our case differs in some respects [15]. The hardware (i.e. the machine environment) is not static even for a single ‘executing’ environment. The machinery and sensory equipment may change over time or differ slightly between sites, although the functionality for the end user is identical. A changing abstract machine means that the engineers must change the behavioural, control level specification to reflect the new (or different) environment. The task for the machine is constant, however, and should be portable to the new executing machine. Therefore a good programming environment must be pro-

vided not only for the task level programming, but also for the control level programming.

THE ARAMIS MODEL

Layered architectures and ARAMIS

The system has been designed with a layered architecture, based on different levels of abstraction. It consists of three different levels: the task programming level, the control level, and the physical level (fig. 1). At the task level the operator specifies what operations should be performed in the physical environment and under what conditions, using a graphical hybrid, rule-based language [7]. The task program executes by setting reference values for the objects in the world model. The control level is responsible for keeping the real world in a state represented in the model of the world (WM), i.e. a servomechanism, as the WM is changed by task program execution. The programming at this level is typically done by control engineers. The physical level is the actual connection to the real world, where explicit I/O is performed with sensors and actuators.

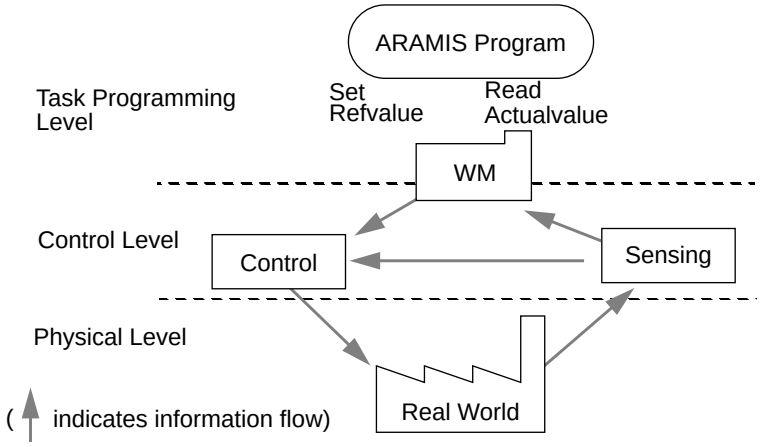


Figure 1 The abstract machine/world model viewed as a reference value that is set, used by a servo controller, controlling the 'real world' and the reading of actual values.

Other principles for layered architectures in this area have been proposed, for example, Brooks [1] defined "behavioural decomposition" as a design criterion for a layered architecture for sensor/actuator control systems, to make them robust and flexible. The different levels then reflects different levels of "competence". Our approach differs from this in that our "modules", the WM-objects (explained below), are not ordered in a simple subsumption hierarchy. Instead each module or object has responsibility for controlling the behaviour of a part of the system. The interaction between the objects is orchestrated by

the task-level program. Each goal (following Brooks) is represented as a ”worker” (or process) on the task-level.

In [4] a three-layer architecture consisting of an analysis layer, a rule layer and a process layer, is presented. In the analysis layer, requested goals or tasks, expressed in temporal logic, are translated into plans for the rule layer to execute. The rule layer transforms the plans into sets of rules which are passed to the process layer. The process layer uses the rules to control the execution of the actions in the plan. Our task level corresponds to the rule layer, but also incorporates the ability for context dependent plan selection. We furthermore model the world as a set of objects which introduces structure into the (process) state representation.

CHIMERA II [13] is a programming environment and operating system designed to reduce the development time for sensor-based control applications. This design is influenced by the NASREM Model for Telerobot Control System Architecture. Chimera II is a real-time operating system which supports multiprocessor environments, but does not provide any specific modelling tools other than light-weight tasks, locally shared memory, local semaphores etc. Similar approaches but incorporating object-oriented paradigms are presented in [10] and [11]. Our work does not focus on the OS support, but rather on the specification tools. The approaches above normally assume application programming in C or C++, which correspond to the control level in our model. Since our work started with investigating the desirable properties of the task-level language, the stress on full object-orientation in the control-level has been decreased.

The task-level programming problem has been thoroughly investigated, e.g. [3][6][9][12]. In our opinion these approaches handle the problem as a conventional computer programming issue. Our model takes into consideration that the executing machine is not discrete and is changing (sometimes during execution) and has a partially unknown state. Even if theoretical work has been applied to these issues, the combination of control algorithms and symbolic plans remains fairly uninvestigated experimentally.

Generally we aim at providing “adequate tools to design and specify intelligent behaviours” for a sensor-based manufacturing system. The assumption is that these kind of processes are well known by the operators and that the processes should be deterministic from a global perspective. Therefore we do not need, or even wish, “exploring” or other non-deterministic behaviour embedded in the system, as in [1]. Another issue is that we aim at different user categories which includes non-programmers for the application programming. This means, for example, that a rule layer as in [4], which does not give a good overview of the task plans, is unsuitable. As plans somehow have to be specified for the system, we prefer a graphical language for representing them.

The world model

The task program operates, for an external observer, by executing actions in the physical environment. The state of the environment is represented in a world model (WM). Each primitive action corresponds to a change of model state.

Each machine or part is viewed abstractly as a modified deterministic finite state automaton (DFA), and is referred to as a world object (WO). The WO state is from the task program view a mapping from WO state variable names to other WO's or to simple data types like integers or symbols (an ordered sequence of symbols is called a symbolic state vector).

Each WO state is identified by the set of {state-variable,value} pairs of the WO. At any time, the actual WO state might be fully or partially known. Partially means that the value of some variables are unknown and impossible to determine by sensory action. A set of WOs corresponds to the abstract machine mentioned previously and a primitive operation by the task program corresponds to changing one (or several, simultaneously) of the state variables of one WO, called a WO transition.

The state of a WO is considered constant unless a transition takes place. A transition occurs in finite, non-zero time. During the transition the values of the state variables which are requested to change are unknown, (in fact known to be changing from an earlier to a later value, unless the variable is explicitly characterized as a sensed variable, see below). Thus, the timeline for the actual WO state consists of intervals of a fully determined constant state, interspersed with time intervals of partially known states during transitions.

Modelling of objects

In the control level there is a numerical model of object behaviour, the control algorithm, which calculates output signals from given input. This numerical model is constant as long as the WO state of the object is constant. If the WO state changes, then the model might or might not change. Let this be an injective mapping from WO state to a numerical model. This means that the set of state variable pairs for the WO denotes a numerical model in the control level and that the model is constant between transitions. Some WO-states have identical control models but with different parameters. Other WO-states differ also in model structure and may therefore be thought of as different modes in the object control model.

Each state variable is represented by two values, one called the reference value and the other called the actual value. Each time the task program orders a WO-transition, the reference value(s) is(are) changed and a request is made to the control level to change the state of the environment to the corresponding state. When this change has been performed, it is acknowledged by the control level.

If the value of a state variable is requested by the task program (possibly in another concurrent sequence of events), the actual value is returned. Normally this value is simply equal to the reference value or unknown. The normal action in this case would be to wait for the completion of the WO-transition, but other options may be possible[1]. However, if the variable is specified at the control

1. The resource allocation problem has not been addressed so far, and must currently be taken care of explicitly by the user/programmer.

level as a sensed variable, it can be calculated (from real sensor data) at any time, even during transitions ordered by the task program. For the task program there is no semantic difference between reading a sensed or an unsensed state variable. This effectively makes it possible to isolate the task program from the configuration, e.g. to selectively simulate sensors or to change the sensor configuration. The world model is discussed more fully in [8].

The control level

The control level must perform actions on the real world so that the difference between the reference value and the actual value of state variables in the world model disappears. This can be done in several ways - a single serial message output on a communication link, setting bits on output ports or by executing a control algorithm which reads sensor data and outputs control signals in a cycle. Such an execution (a transition, servoing algorithm) will bring the real object into the state desired by the task program. On the other hand, sometimes the new state is not an equilibrium state, thus there is a need to execute maintaining control algorithms (these would correspond to control modes for the object in question). The transition algorithms are parameterized by the start and end value of the state variable(s) in the transition and the maintaining algorithms are, of course, parameterized by the end value of the last transition.

The control level must also transfer sensed information back to the world model. This is, however, only necessary in those cases where the intention is for the task program to be able to specify a feedback loop. For mixed sensors/actuators, i.e. a sensor which has some controllable feature (like the direction of a supervision video camera), there must never be any possibility for interaction between feedback loops defined in the task program and the transition and maintaining algorithms defined for the corresponding WO. This cannot be guaranteed in the present model, but is the responsibility of the programmer of the control level.

THE IMPLEMENTATION OF THE CONTROL LEVEL

The ARAMIS system consists of three levels, the task programming level, the control level and the physical level (fig. 2). In the task programming level the system operator can write task programs. The ARAMIS programs are specified in a graphical editor and are stored in a memory resident database, ITEM. When modelling objects of the manufacturing system in the control level, which we hereafter call *active components*, an object-oriented view is used. Component code (control algorithms) is specified in a special programming language that is compiled and downloaded to a real-time kernel. Component behaviour is specified by DFA[1]-diagrams, for each component, that specifies the algorithms to execute in case of a change in the WM. Component behaviour and component code are specified in an integrated component editor/compiler tool. The real-time kernel is implemented in a multi-tasking Forth system exe-

1. Deterministic Finite Automation

cuting on a separate real-time processor which directly communicates with the actual sensors and actuators.

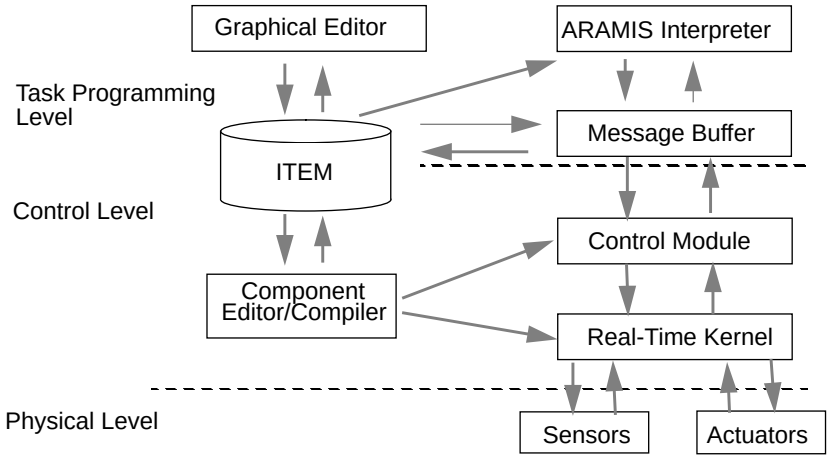


Figure 2 Modules and tools in the system implementation

Modelling the world as components

Components are divided into two main hierarchies, active and passive components (fig. 3). An active component is a model of a physical object in the world (e.g. a sensor or an actuator) or a virtual object (e.g. an abstract sensor using sensor fusion of several physical sensors). Active components have a local state associated with the control algorithms and are represented as concurrent processes. The hierarchy of active components specifies the levels of parentage or kinship (e.g. an asynchronous- and a synchronous motor can both be seen as instances of an electrical motor). The inheritance from a parent component includes both the static code and the dynamic behaviour of the parent. All objects in the WM are modelled as active components, but not all active components are WM-objects.

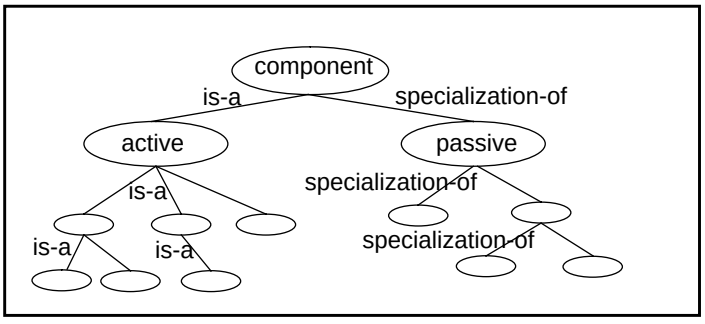


Figure 3 The component hierarchy

Active components (fig. 4) may be structured to consist of other active components to model complex behaviour and may use passive components to structure and share code with other components. Active components which are used by others cannot be referenced from the task level. Active components which are not used by other components are called top components and can be referenced from the task program.

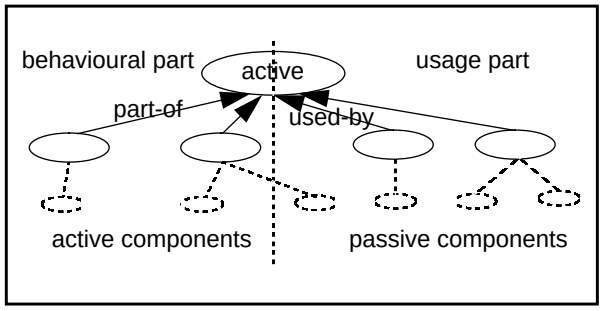


Figure 4 The active component aggregation model

A passive component is a package (in a software module sense) which contains related code (e.g. a package containing mathematical functions). Passive components have no local state and are represented as a collection of data and code. The hierarchy of passive components specifies the levels of abstraction (e.g. packages for path-following algorithms with and without sensory feedback can both be seen as specializations of an abstract path following package). By having the same external interface in an abstract passive component, no changes have to be made in an active component that uses the passive component, when one algorithm is exchanged with another (procedural abstraction).

Calls to active components are implemented as implicit message passing (i.e. syntactically as procedure calls, but semantically as interprocess calls) and calls to passive components are implemented as regular procedure calls (both syntactically and semantically). Passive components can use other passive components. Passive components can not define or reference local state variables since this would violate the integrity (OO encapsulation) of active components, according to the ARAMIS model.

The component description language

The component description language consists of two parts, an algorithm specification and a behaviour specification. Algorithms are written as functions and procedures in a typed, imperative language. The language includes primitive types like booleans, integers and reals together with traditional logic and arithmetic operators. The language includes traditional control constructs such as *if-then-else* and *case* constructs and iterative constructs such as *for*, *repeat* and *while* loops, and recursion. The language also includes primitive I/O for communication with sensors and actuators.

The behaviour specification is a state transition graph (or a state machine), where the states denotes a set of legal valuesets of the WO state variables and the transitions represent legal changes of the WO state variables. The state transitions denotes an initial and a final state, an algorithm (in the algorithm specification) which will execute the state change and a time constraint specifying a possible time out for the state change. Maintenance algorithms can also be specified for each state. The state transition graph is made to cover all remaining (illegal) subsets of the value domain by adding one or several error states.

Component definitions are compiled and downloaded to the real-time kernel in two phases. In the first phase (the instantiation phase) the algorithm specification of the component is compiled to target code. In the second phase (the installation phase) the behaviour specification is used to install the active component as an executable unit, by creating a state table in the control module reacting to changes in the WM and a task in the real-time kernel which can execute the changes in the real world.

To be more specific, an instantiated active component consists of two main parts:

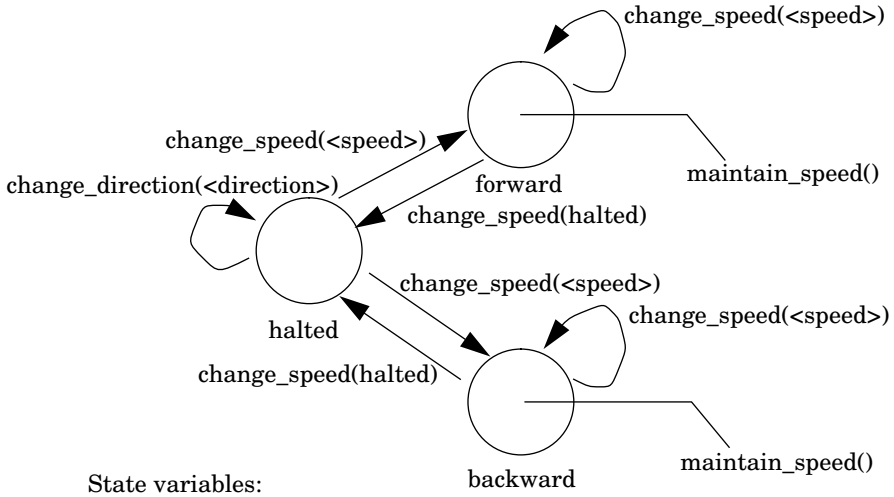
- A database part, stored in a memory resident database, consisting of one static and one dynamic part:
  The static part includes the complete description, i.e. both the algorithm specifications and the behaviour specifications. The dynamic part includes state variables, accessible from the task program through the memory resident database, and a state table generated at the instantiation from the behaviour specification (top components only). State variables are represented as a tuple with an actual value and a reference value. When a state variable of an active component is changed by the task program, the reference value is set. The state table is then indexed by the current state of the component, returning a function that takes the reference state vector (the values of all the state variables of the component) and returns the next state, the transition algorithm to be called, and the maintenance algorithm of the next state. When the state transition is acknowledged from the real-time kernel, i.e. when the transition algorithm has terminated successfully, the actual value of the state variable is updated. When the task program reads a state variable, the new actual value is returned. The actual value might also be changed due to asynchronous changes in the real-time kernel induced by nondeterministic events in the environment. In this case the actual value is changed directly. The first class of state variables are called implicitly determined and the latter are explicitly determined (sensed).

- A real-time kernel part consisting of:
  One main process representing the active top component:
  - A communication channel.
  - A table of constants and state variables.
  - A table of algorithms.
  - A table of subcomponents represented as processes defined as above.

Installed components are registered in the control level and are activated in a system initialization phase. When an active component is declared to consist of other active components, the subcomponents are also instantiated and installed before the main component. An active component class can be instantiated both as a stand-alone component and as being part of another component (i.e. as two different instances), but only the stand-alone component will have a dynamic part installed in the data base and thus be visible from the task level.

A COMPONENT EXAMPLE

We illustrate the modelling of component behaviour with an example - a conveyer belt. The model consists of an active component consisting of a motor, a speed sensor and the actual conveyer belt. The main component may be modelled as follows. The conveyer belt can be in three different states, halted, moving forward or moving backwards (fig. 5).



State variables:
direction (forward,backward)
speed (halted,low,medium,high)

Algorithms:
change_direction(direction)
change_speed(speed)
maintain_speed()

State constraints:
halted: speed = halted
forward: direction = forward, speed = low,medium,high
backward: direction = backward, speed = low,medium,high

Figure 5 Example of a state graph - a conveyor belt

The component has two WO state variables, direction and speed, which can

be directly manipulated by the task level. The halted state is the initial state. The initial value of direction in the halted state is unknown. The algorithm change_direction will be called in the real-time kernel to change the direction. The algorithm change_speed is called to set the speed to either halted, low, medium or high and, if the algorithm succeeds, a state change will occur. In the states forward and backward the maintenance algorithm maintain_speed is executed continuously to make sure the speed is kept. When the speed is changed, the maintenance algorithm is interrupted and the algorithm change_speed is called. If the speed is changed to halted, the state changes to halted. If the speed is changed to some other value than halted, the current state will be kept and the maintenance algorithm will be called again after the speed change. If either a transition algorithm or a maintenance algorithm fails, the task level is notified by an error message. In the case where an inconsistent state occurs, i.e. if the state variables are set to values not consistent with any of the defined states, a transition to an error state will occur and the task level will be notified. In the example, this is the case if direction is unknown when setting speed to some other value than halted. How the values of the WO state variables, seen by the task level, correspond to the values in the real-time kernel has to be defined in a mapping between the values of the variables in the WM and the "real" values of the variables in the real-time kernel. This mapping defines the accuracy to which the maintenance algorithms should maintain the variable values and it also defines the intervals by which asynchronous changes of values (from sensory data) should be reported to the WM.

CURRENT STATUS

Hardware and Software Platforms

The hardware for the task level and the control level programming is a SUN SPARCstation 1. For the real-time environment a VME-bus based Motorola 68020 is used. The software for the SPARC is the XEROX EnVõs Common Lisp environment and for the Motorola, the TILE multi-tasking Forth environment[1].

The Real-Time Environment

The real-time environment consists at present of a robot with a three-finger gripper and some various sensors, controlled by the Motorola 68020. The robot is a conventional 6-axis PUMA 560. Some additions have been made to the control system to provide extended external control. The gripper is specially designed to be more flexible and controllable than conventional grippers, and is also prepared for mounting sensors onto it. A CCD camera is used as vision sensor, providing 256x256 pixels resolution and placed as a scene camera. The vision system is situated on a separate computer, communicating with the VME computer. Tactile sensor arrays are mounted on the gripper's fingers, giving

1. M. Patel, Dept. Computer and Information Science, Linköping University.

imprint patterns. An ultra-sonic sensor on the gripper can be used as range/ proximity sensor. A strain-gauge sensor on the robot arm can be used as force sensor. A similar sensor is also intended to be mounted in the gripper, to be able to measure the grip force.

For accessing and controlling sensors and actuators from the Forth environment, some extensions have been made. These include code for reading and writing from analogue and digital I/O boards and communicating with the robot control system and the vision computer. The communication is performed by using a simple package based protocol that is embedded in the communication code, thus making it transparent to the user.

FUTURE WORK

Many aspects of the architecture described in this paper needs to be expanded further. Some of the urgent extensions include fault handling and the scheduling of processes in the real-time kernel. Simple fault handling is supported at present, but this does not include propagation and translation of error information between the two levels in the architecture. Error information at the control level needs to be translated using the context of the task being executed at the task level. This involves translating from how the error presents itself to what the logical cause of the error might be. By extending control level programming to include components as first-class objects, e.g. by moving closer to an object-oriented programming language, it will be possible to implement generic control algorithms. Today every executable algorithm must be encapsulated within the corresponding component. Other areas that need further work include support for software requirement analysis to determine the consistency of a component behaviour specification and support for the distribution of data and programs on distributed hardware.

ACKNOWLEDGEMENTS

We would like to thank NUTEK, the Swedish National Board for Industrial and Technical Development, and CENIIT at Linköping University, for funding this project. We would also like to thank other members of the Measurement Technology group at IFM for providing us with a real-time environment and for teaching us more about sensor technology.

REFERENCES

[1] R. A. Brooks, “A Robust Layered Control System for a Mobile Robot”, in *IEEE Journal of Robotics and Automation* vol. RA-2 No. 1, March 1986.

[2] J. J. Clark and A. L. Yuille, “Data Fusion for Sensory Information Processing Systems”, Kluwer Academic Publ.

[3] G. C. Gini and M. L. Gini, ”Dealing with World-Model-Based Programs”, *ACM Transactions on Programming Languages and Systems*,7,2, 1985.

[4] J. Hultman, A. Nyberg, and M. Svensson, “A Software Architecture for Autonomous Systems”, *6th International Symposium on Unmanned, Untethered Submersible Technology*, Elicott City, Maryland, 1989.

[5] S. Levi and A. K. Agrawala, “Real-Time System Design”, chap. 7, McGraw-Hill, 1990.

[6] L. I. Lieberman and M.A.Wesley, “AUTOPASS: an automatic programming system for computer controlled mechanical assembly”, *IBM J. Research & Development*, 21,4, 1977.

[7] P. Loborg and A. Törne, “A Hybrid Language for the Control of Multimachine Environments”, in *Proc. of 4th International Conference on Industrial & Engineering Applications of AI & Expert Systems.*, Koloa, Hawaii, 1991, by University of Tennessee Space Institute.

[8] P. Loborg, M. Sköld, and A. Törne, “A Hierarchical Software Architecture for Control of Industrial Robots and Manufacturing Equipment”, presented at 1st National Symposium on Real-Time Systems, Proc. in Tech. Rep. No. 30, Dept. of Computer Systems, Uppsala University, June 1991.
CAELAB Memo 92-01.

[9] T. Lozano-Perez and P.H. Winston, “LAMA: a language for automatic mechanical assembly”, in *Proceedings of the 5th IJCAI Conference*, 1977.

[10] C. W. Mercer and H. Tokuda, “The ARTS Real-Time Object Model”, in *Proceedings of 11th IEEE Real-Time Systems Symposium*. December, 1990.

[11] D. J. Miller and R. C. Lennox, “An object-oriented Environment for Robot System Architectures”, in *Proceedings of the IEEE International Conference on Robotics and Automation in Ohio 1990*, Vol. 1, Page 352.

[12] B. Shepherd, “Task-level programming of a robot using an intelligent human-robot interface”, in the *Proceedings of the 2nd International Conference on IEA-AIE*, ACM, 1989.

[13] D. B. Stewart, D. E. Schmitz, and P. K. Khosla, “Implementing Real-Time Robotic Systems Using CHIMERA II”, in the *Proceedings of the IEEE International Conference on Robotics and Automation in Ohio 1990*, vol. 1, Page 598.

[14] A. Törne, “The Instruction and Control of Multi-Machine Environments”, in *Proceedings of the 5th International Conference on Applications of Artificial Intelligence in Engineering*, Springer-Verlag, Vol. 2, Page 137, Boston, July 1990.

[15] H. Van Dyke Parunak, J. Kindrick, and B. W. Irish, “Viewing the factory as a Computer: A Cognitive Approach to Materials Handling”, in *Artificial Intelligence - Manufacturing Theory and Practice*, eds. S.T. Kumara, R. L. Kashyap, A.L. Soyster, The Institute of Industrial Engineers, Industrial and Management Press, 1989, Pages 225-264.

15.2 Paper II

T. Risch and M. Sköld: Active Rules based on Object-Oriented Queries, in special issue on Active Databases in the Data Engineering Bulletin 15(1-4), Pages 27-30, 1992.

Active Rules based on Object-Oriented Queries

Tore Risch
torri@ida.liu.se

Martin Sköld
marsk@ida.liu.se

Department of Computer and Information Science
Linköping University
Sweden

Abstract

We present a next generation object-oriented database with active properties by introducing rules into OSQL, an Object-Oriented Query Language. The rules are defined as Condition Action CA rules and can be parameterized, overloaded, and generic. The condition part of a rule is defined as a declarative OSQL query and the action part as an OSQL procedural body The action part is executed whenever the condition becomes true. The execution of rules is supported by a rule compiler that installs log screening filters and uses incremental evaluation of the condition part. The execution of the action part is performed in a check phase that can be carried out after any OSQL commands in a transaction or at the end of the transaction. Rules are first-class objects in the database which makes it possible to make queries over rules. We present some examples of rules in OSQL, some implementation issues, some expected results, and some future work such as temporal queries and real-time support

Key Words: Active Database, Object-Oriented Query Language, Object-Oriented Rules

1 Introduction

A powerful query language will be an essential part of the next generation Object-Oriented (OO) database systems. When active properties are introduced into these databases, the query language should be extended to support them.

The HiPac [4] project introduced *ECA rules* (Event-Condition-Action). The Event specifies when a rule should be triggered. The Condition is a query that is evaluated when the Event occurs. The Action is executed when the Event occurs and the Condition is satisfied.

In Ariel [6] the Event was made optional, making it possible to specify *CA rules*, which use only the Condition to specify *logical events* which trigger rules. Rules in OPS5 [1] and monitors in [8] have similar semantics. In ECA rules the user has to specify all the relevant *physical events* in the Event part. We believe that CA rules are more suitable for integration in a query language, since they are more declarative. CA rules make physical events implicit, just as a query language makes database

navigation implicit.

We define active rules by extending the OO query language OSQL of Iris [5]. OSQL is based on functions for associating attributes with objects (both stored and derived). OSQL permits functional overloading on types, and types and functions are first-class objects. Likewise, rules are first-class objects in the database too [3]. This makes it possible, for example, to make queries over rules. By implementing rules on top of OSQL, overloaded and generic rules are possible, i.e. rules that are parameterized and that can be instantiated for different types. We also utilize the optimizations performed by the OSQL compiler [7].

Each rule is defined by a <Condition,Action> pair, where the Condition is a declarative OSQL query and where the Action is an OSQL database procedure body. The rule language thus permits CA rules where the Action is executed (i.e. the rule is triggered) whenever the Condition becomes true, similar to OPS5 and Ariel. Unlike those systems, the Condition can refer to derived functions (which correspond to views). Data can be passed from the Condition to the Action of each rule by using shared query variables. By quantifying query variables set-oriented Action execution is possible [11].

We are implementing our ideas in the research prototype, AMOS[1] (Active Mediators Object System), by extending a Main-Memory version of Iris, WS-Iris[7]. OSQL queries are compiled into execution plans in an OO logical language. The system logs all side effect operations on the database. The rule compiler analyzes the execution plan for the Condition of each rule. It then generates 'log screening filters' which check each event that is added to the log. When a log event passes a log screening filter associated with a Condition, it indicates that the event can cause the corresponding rule to fire. The screening of the log is often complemented with incremental evaluation [9],[10] of the Condition.

Distributed execution of AMOS is being implemented too, and we plan to introduce temporal queries and real-time facilities as well.

2 Object-Oriented Query Rules

In AMOS OSQL has been extended with rules having a syntax conforming to that of OSQL functions as closely as possible. AMOS supports rules of CA type where the Condition is an OSQL query, and the Action an OSQL procedure body. The syntax for rules is the following:

create rule rule-name param-spec **as**
  when [*for-each-clause* | *predicate-expression*]
  do [**once**] *action*
where
for-each-clause ::=

1. The AMOS project is supported by Nutek (The Swedish National Board for Industrial and Technical Development) and CENIT (The Center for Industrial Information Technology), Linköping University

for each *variable-declaration-commalist* **where** *predicate-expression*

The *predicate-expression* can contain any boolean expression, including conjunction, disjunction and negation. Rules are activated and deactivated by:

activate rule-name ([*parameter-value-commalist*])

deactivate rule-name ([*parameter-value-commalist*])

The semantics of a rule are as follows: If an event in the database changes the boolean value of the condition from *false* to *true*, then the rule is marked as *triggered*. If something happens later in the transaction which causes the condition to become false again, the rule is no longer triggered. This ensures that we only react to logical events[1]. In the *check phase* (usually done before committing the transaction), the actions are executed of those rules that are marked as triggered. If an action is to be executed only once per activation, the rule is deactivated after the action has been executed. We can also introduce an *immediate* coupling mode [4] by instructing the system that the check phase is to be done immediately after each OSQL command.

Example 1:

The salary changes of employees and managers are to be monitored. We want to ensure that only managers can have their salaries reduced. First we define the employee and manager types and the respective income functions, where managers receive an additional bonus:

```
create type person;
create type employee subtype of person;
create type manager subtype of employee;
create function name(person) -> charstring as stored;
create function mgrbonus(manager) -> integer as stored;
create function income(employee) -> integer as stored;
create function income(manager m) -> integer i
  as select i where i = employee.income(m) + mgrbonus(m);
create employee(name,income) instances
  :joe ('Joe Smith',30000);
create manager(name,employee.income) instances
  :harold ('Harold Olsen',80000);
set mgrbonus(:harold) = 10000;
```

Then we define procedures for what to do when a salary is decreased:

```
create procedure compensate(employee e)
  /* employee income cannot be decreased */
  as set income(e) = previous income(e);
create procedure compensate(manager);
/* dummy procedure, managers are not compensated */
```

1. To support physical events the system should provide functions that change values whenever a physical event occurs and thus can be referenced in the condition of a rule.

The function `compensate` uses the system operator `previous` to fetch the value of a function at the previous checkpoint.

Finally we define the rule to detect decreasing salaries for all employees:

```
create rule no_decrease() as
  when for each employee e
  where income(e) < previous income(e)
  do compensate(e);
```

Activate the rule:

```
activate no_decrease();
```

If an employee who is not a manager has his salary decreased, the rule will automatically set the salary back to the old value at check time:

```
set income(:joe) = 20000;
/* => reset income(:joe) to 30000 at check time */
```

Note: Since the rule is defined for all employees, and manager is a subtype of employee, the rule is overloaded for managers (because the functions `income` and the procedure `compensate` are overloaded). If a person of type manager gets a salary reduction, no action is taken. This is an example of a set-oriented rule. The action is executed for every binding of the universally quantified variable `e` for which the condition is true.

Example 2:

Rules can be parameterized and instantiated with different arguments.Take a rule that ensures that a specific employee has an income below a certain maximum income, and the transaction is rolled back if an employee receives an income above the threshold. This maximum income is fixed for all employees, but can vary for individual managers.

```
create function maxincome(employee) -> integer
  as select 50000;
create function maxincome(manager) -> integer as stored;
create rule exceeding_maxincome(employee e) as
  when income(e) > maxincome(e)
  do rollback;
```

Set the income limit for Harold:

```
set maxincome(:harold) = 120000;
```

Activate the rule for a particular employee Joe and manager Harold:

```
activate exceeding_maxincome(:joe);
activate exceeding_maxincome(:harold);
set income(:joe) = 75000;
/* rollback at check time because 75000 > 50000 */
set maxincome(:harold) = 90000;
/* rollback at check time because 90000+10000 > 90000 */
set mgrbonus(:harold) = 45000;
/* rollback at check time because 80000+45000 > 120000 */
```

It is non-trivial to determine the physical events that trigger an OSQL rule with many interdependent and overloaded functions, such as the rule above. Hence we let the compiler determine this. This illustrates the convenience of CA rules.

Example 3:

Since types are first class objects, one can write generic rules that are instantiated for a specific object type:

```
create rule exceeding_maxincome(type t) as
  when for each employee e
  where typesof(e) = t and
  income(e) > maxincome(e)
  do rollback;
```

Activate the rule for all managers:

```
activate exceeding_maxincome(typenamed('manager'));
```

Since rules are first-class objects in the database, one can make queries over rules. For example, the system could provide a function that returns all active rules dependent on a certain object type or a function that takes a rule as argument and returns all the functions it depends on.

3 Expected results

The extension of OSQL with rules is expected to provide a powerful language to express active properties in an object-oriented database. The overloading of rules provides a way to specify reusable rules that can be applied uniformly in different situations. One of the goals in the project is to investigate if CA rules can be implemented as efficiently as ECA rules. This involves efficient event detection as well as incremental evaluation of rule conditions. We will verify the applicability of OO rules by investigating how they can be used for various applications, e.g. CIM.

4 Future work

Temporal rules can be introduced by having functions that vary over time and by time-stamping events in the database. The condition can then refer to the time when a certain event occurred. By introducing a timer event, a rule can be triggered at a certain time. These extensions do not support all the possible reasoning that can be made in an event algebra such as [2]. However, it allows for reasoning about whether one event happened before another or vice versa (by comparing time-stamps).

Introducing real-time properties in the database would require taking the cost of executing an action into account. Active database facilities are important for real-time applications that, e.g., monitor combinations of sensor data and perform actions whenever 'interesting' situations occur. The rule language will need to be

complemented with timeliness constraints for rule conditions and actions.

References

[1] Brownston L., Farell R., Kant E., and Martin A.: Programming Expert Systems in OPS5, *Addison-Wesley, Reading Mass.*, 1986.

[2] Chakravarthy S. and Mishra D.: An Event Specification Language (Snoop) for Active Databases and its Detection, *UF-CIS Technical Report*, TR-91-23, Sept. 1991.

[3] Dayal U., Buchman A.P., and McCarthy D.R.: Rules are objects too: A Knowledge Model for an Active, Object-Oriented Database System, *Proc. 2nd Intl. Workshop on Object-Oriented Database System*s, Lecture Notes in Computer Science 334, Springer-Verlag, 1988.

[4] Dayal U. and McCarthy D.: The architecture of an Active Database Management System, *ACM SIGMOD*, 1989, Pages 215-224.

[5] Fishman D. et. al.: Overview of the Iris DBMS, *Object-Oriented Concepts, Databases, and Applications*, ACM press, Addison-Wesley Publ. Comp., 1989.

[6] Hanson E. N.: Rule Condition Testing and Action Execution in Ariel, *ACM SIGMOD*, 1992, Pages 49-58.

[7] Litwin W. and Risch T.: Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 6, Dec. 1992.

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

[9] Rosenthal A., Chakravarthy S, Blaustein B., and Blakely J.: Situation Monitoring for Active Databases, in *the Proceedings of the VLDB Conference*, Amsterdam, 1989.

[10] Paige R. and Koenig S.: Finite Differencing of computable expressions, in *ACM Transactions on Programming Languages and Systems*, 4.3, July 1982, Pages 402-454.

[11] Widom J. and Finkelstein S.J.: Set-oriented production rules in relational database system, in *the Proceedings of the ACM SIGMOD International Conference on Management of Data*, Atlantic City, New Jersey, 1990, Pages 259-270.

15.3 Paper III

G. Fahl, T. Risch, and M. Sköld: AMOS - An Architecture for Active Mediators, in Proceedings of the Workshop on Next Generation Information Technologies and Systems (NGITS'92), Haifa, Israel, June 1993.

AMOS - An Architecture for Active Mediators

Gustav Fahl, Tore Risch, Martin Sköld

Department of Computer and Information Science
Linköping University
S-581 83 Linköping, Sweden
E-mail: gusfa@ida.liu.se, torri@ida.liu.se, marsk@ida.liu.se

Abstract

AMOS[1] (Active Mediators Object System) is an architecture to model, locate, search, combine, update, and monitor data in information systems with many workstations connected using fast communication networks. The approach is called *active mediators*, since it introduces an intermediate level of 'mediator' software between data sources and their use in applications and by users, and since it supports 'active' database facilities. A central part of AMOS is an Object-Oriented (OO) query language with OO abstractions and declarative queries. The language is extensible to allow for easy integration with other systems. This allows for knowledge, now hidden within application programs as local data structures, to be extracted and stored in AMOS modules. A distributed AMOS architecture is being developed where several AMOS servers communicate, and where queries in a multi-database language are allowed to refer to several AMOS databases or other data sources. An overview is provided of the architecture and components of AMOS, with references to ongoing and planned work.

1 Introduction

Future computer supported engineering, manufacturing, and telecom environments [Lob93, Imi92] will have large number of workstations connected with fast communication networks. Workstations will have their own powerful computation capacities which store, maintain, and make inferences over local engineering data- and knowledge-bases, or *information bases*. Each information base is maintained locally by some human operator and is autonomous from other information bases. Each information base will need a set of DBMS capabilities, e.g. data storage, a data model, a query and data modelling language, transactions, and external interfaces. The classical relational database languages are not powerful enough for the manipulations needed, for example, to build advanced models to filter and extract required informa-

1. The AMOS project is supported by TFR (The Swedish Research Council for the Engineering Sciences), NUTEK (The Swedish National Board for Industrial and Technical Development) and CENIIT (The Center for Industrial Information Technology), Linköping University.

tion. Facilities are also needed to support 'reactive' applications that sense changes in information, i.e. *active* database facilities [DE92].

The AMOS (Active Mediators Object System) architecture uses the *mediator* approach [Wie92] that introduces an intermediate level of software between databases and their use in applications and by users. We call our class of intermediate modules *active mediators*, since our mediators support active database facilities.

We have identified four classes of mediators needed in our architecture, which will be explained in more detail in the next sections:

1. *Integrators* that retrieve, translate, and combine data from data sources with different data representations.
2. *Monitor models* that notify mediators or application programs when data updates of interest occur.
3. *Domain models* that represent application-oriented models and database operators.
4. *Locators* that locate mediators and data in a network of AMOS servers.

The AMOS architecture is built around a main memory based platform for intercommunicating information bases. Each AMOS server has DBMS facilities, such as a local database, a data dictionary, a query processor, transaction processing, and remote access to data sources. Central to the AMOS architecture is an OO query language, AMOSQL, which is a derivative of OSQL [Fis89]. AMOSQL supports OO abstractions and declarative queries. It is extensible to allow for easy integration with other systems. AMOS makes it possible to extract knowledge that currently is hidden within application programs as local data structures and represent it in AMOS modules. Query processing must be efficient enough to encourage the use of local embedded databases linked into applications without significant performance penalty. The query and modelling language must also be powerful enough to store complex knowledge models. Furthermore, queries should be allowed to access more than one autonomous AMOS server as well as other data sources. It should be possible to state queries using the same multi-database query language independently of where the queried data reside. AMOSQL also supports *active rules* [Ris92] that execute when certain more or less complex conditions change.

To support the initial work on AMOS, a main-memory OO DBMS engine, WS-IRIS [Lit92], is being modified. It provides an extended OSQL version and fast execution. WS-IRIS is open and easy to modify for research purposes. The system supports extensibility through *foreign functions* written in an external programming language (usually Lisp or C). A query optimizer translates OSQL queries and methods into optimized execution plans in an internal logical language, ObjectLog [Lit92]. The optimizer is extensible so that *cost hints* can be associated with arbitrary OSQL functions to guide the optimizer about alternative execution plans. We are developing new optimization strategies by new kinds of transformations on the ObjectLog query plans.

2 AMOS Components

Figure 1 illustrates how a set of application programs access a set of data sources through active mediators. An overview follows of the work we are doing on each kind of mediator.

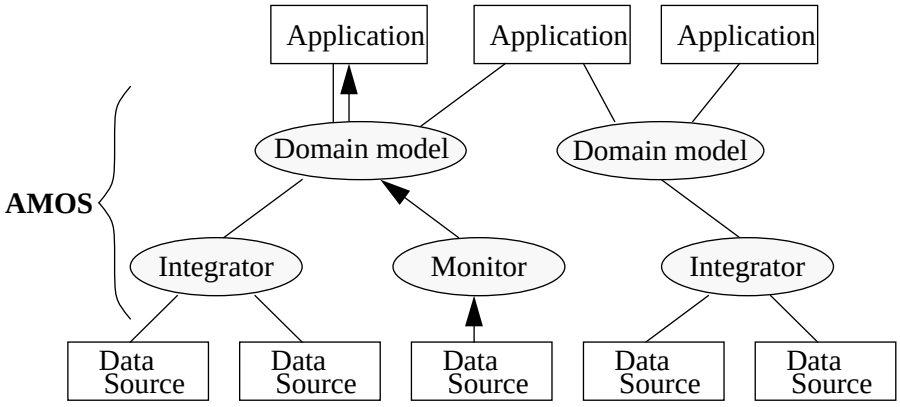


Figure 1 Active mediators of different classes mediating between data sources and users/applications

2.1 Integrators

Data sources are likely to be heterogeneous. Data could be stored in different DBMSs, using different data models. Even if the same DBMS is used, data could still be semantically heterogeneous [She91].

Integrators are responsible for making this heterogeneity transparent to higher-level mediators and applications. Integrators retrieve and combine data from underlying data sources, giving applications and higher-level mediators an integrated view of data and decoupling them from the necessity of understanding multiple data models.

Integrators are implemented with two kinds of AMOS servers; *Translation AMOS* (TAMOS) servers and *Integration AMOS* (IAMOS) servers (see figure 2). We will initially concentrate on *access* to heterogeneous data sources, not updates.

Much current research is being applied to heterogeneous database systems [She90]. The usual way to deal with data model heterogeneity is to map the schemas of the data sources to schemas in a common data model (CDM). In most previous research a relational CDM has been used. This is inadequate if there are data sources with a data model that is semantically richer than the relational model. In these cases, it will not be possible to capture all of the semantics of the data sources in the CDM. Ideally, the expressiveness of the CDM should be greater than, or equal to, the expressiveness of all the data models of the data sources. We use the functional and object-oriented data model from IRIS [Fis89] as our CDM.

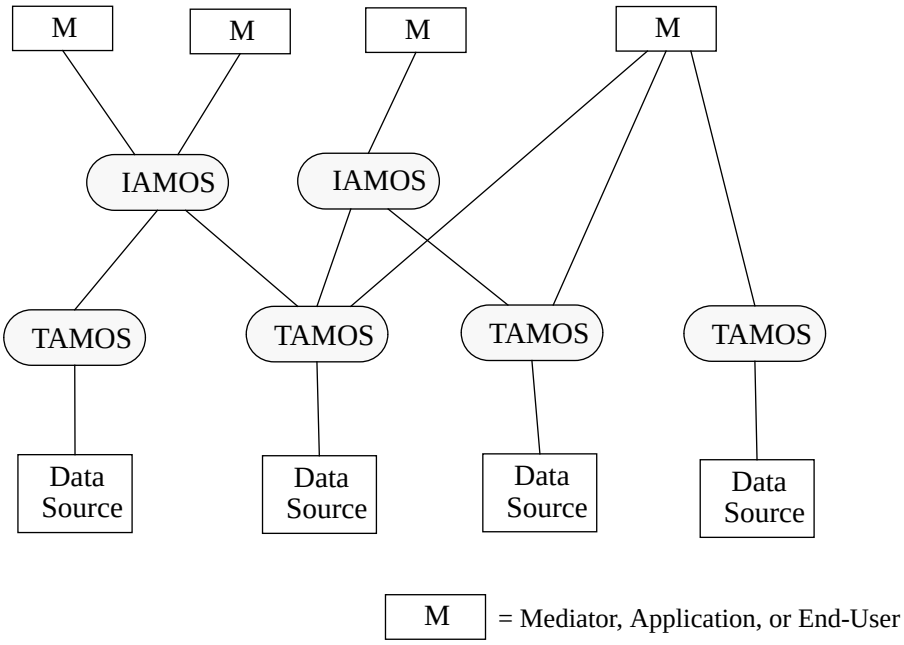


Figure 2 Translation AMOS (TAMOS) and Integration AMOS (IAMOS) - the servers implementing Integrators

Related work of particular interest include the Multibase [Lan82] and Pegasus [Ahm91] projects.

Multibase has a similar architecture and uses a functional data model as their CDM and DAPLEX [Shi81] as the Data Manipulation Language (DML). AMOSQL is a DAPLEX derivative, but an important difference is that AMOSQL is object-oriented. Queries in AMOSQL can return OIDs. Another difference is the role of the translation component. This will be discussed in section 2.1.1.

The Pegasus project also uses the IRIS data model as their CDM and an extension to OSQL as the DML. The main difference to AMOS is architectural. A Pegasus server performs both translation and integration, whereas in AMOS this is separated in two modules. There is one type of TAMOS server for each type of data source. Each TAMOS server only needs to know the data model of one data source and how to map this to the CDM. IAMOS servers only need to understand the CDM. The Pegasus server must understand all underlying data models and must have language constructs for mapping each of these data sources to the CDM.

2.1.1 TAMOS

Translation AMOS servers map the schemas of the data sources to schemas in the CDM. There is one TAMOS server for each kind of data source. An AMOSQL query sent to a TAMOS server is translated to calls to the underlying data source. The results of these calls are then processed to form answers to the AMOSQL query.

A TAMOS server can be used by one or more IAMOS servers or directly by applications and other mediators. We are initially developing TAMOS servers for a relational database (SYBASE), and for a conventional file data source.

A central problem is how to get OO access to a non-OO data source. In the method chosen, each TAMOS server will contain descriptions of how to map values in the underlying data source to object identifiers (OIDs). OIDs are dynamically generated when necessary and are thereafter maintained by the TAMOS server.

When the data model of the data source provides fewer semantic modelling constructs than the CDM, mapping a data source schema into a schema in the CDM involves a semantic enrichment process [Cas93]. We want TAMOS to capture as much of the semantics of the data source as possible. This is different from, for instance, Multibase, where the translated schema is the simplest possible and where the semantic enrichment is performed in the integration module. We want to avoid this approach, which leads to increased communication between the translation and integration modules. Our approach makes query optimization in TAMOS more difficult, since query processing involves both calls to the data source and local TAMOS computations. The optimizer must find the most effective combination of these.

TAMOS query plans are represented by an extended version of ObjectLog. Some TAMOS types will have instances corresponding to atomic values in the data source. OIDs must be generated when a query returns objects of such a type. Similarly, OIDs must be converted back to atomic values when they are used in queries to the data source. Thus, TAMOS query plans often contain statements which map between OIDs and atomic values. However, when a query is a used as a subquery of a larger query and thus query plans are combined, the OID mappings on intermediate results are not needed. The optimizer recognizes these cases and removes such unnecessary OID mappings from the execution plan. This makes larger portions of TAMOS execution plans translatable to, e.g., relational queries to the data source, which minimizes communication between TAMOS and its data source.

2.1.2 IAMOS

An Integration AMOS server combines data from other AMOS servers (TAMOS or IAMOS) and presents an integrated view of the data. A query sent to an IAMOS server is transformed into several queries for the underlying AMOS servers. The results of these queries are then processed to form an answer to the initial query. Special optimization techniques are needed compared to conventional distributed DBSs due to the heterogeneity and autonomy of the data sources [Lu93].

To define the mapping between the integrated IAMOS schema and the underlying TAMOS/IAMOS schemas, an OO multi-database query language is needed. This

language is used to define object views [Abi91] in terms of combinations of data from other AMOS servers and from local data and views.

To access the data sources it is not necessary to use an IAMOS server. Queries can be put directly to TAMOS servers using the multi-database language. Using the terminology from [She90], our architecture can be seen as a combination of a Loosely Coupled Federated DBS and a Tightly Coupled Federated DBS (with multiple federations).

Thus, the same OO multi-database query language is used for local queries, multi-database queries, and for defining multi-database object views. One proposal being studied for such a language is found in [Cho92].

2.2 Monitor Models

Some applications require a mechanism to handle the problem of dynamically changing data. Mediators are provided that continuously monitor these data changes and notify applications when changes of interest for some application occur. These *monitor models* allow application programs to cooperate via AMOS. Of particular interest is to provide means to build monitor models that filter change in data sources, so that irrelevant changes are ignored.

To support monitor models AMOS provides active database capabilities by *active rules* using AMOSQL queries [Ris92]. The active database capabilities of AMOS are used also for other purposes than monitor models, such as for consistency checking. AMOSQL permits functional overloading on types, and types and functions are first-class objects. By implementing rules on top of AMOSQL, overloaded and generic rules are possible, i.e. rules that are parameterized and that can be instantiated for different types. We also utilize the optimizations performed by the AMOSQL compiler.

The HiPac [Day89] project introduced active *ECA rules* (Event-Condition-Action). The event specifies when a rule should be triggered. The condition is a query that is evaluated when the event occurs. The action is executed when the event occurs and the condition is satisfied.

In our active rules the event is made optional by defining each rule as a pair <Condition,Action>, where the condition is a declarative OO query and where the action is an OO database procedure body, i.e. a *CA rule*. An action is executed (i.e. the rule is triggered) when the condition becomes true. We believe that CA rules are more suitable for integration in a query language, since they are more declarative. CA rules make physical events implicit, just as a query language makes database navigation implicit. OPS5 [Bro85] and Ariel [Han92] have similar rule semantics. Unlike those systems, the condition in an AMOS rule can refer to derived AMOSQL functions (which correspond to views). Data can be passed from the condition to the action of each rule by using shared query variables. By quantifying query variables set-oriented action execution is possible [Wid90]. Rules are furthermore parameterized and type overloaded, so that they can be instantiated for objects of different types.

An interface has also been developed between active rules and application programs where the programmer can specify *trackers* [Ris89], which are procedures or processes of the application or other AMOS servers that are invoked or called by AMOS

when a rule action is triggered. AMOS thus needs a callback mechanism that is invoked from active rules. Such a mechanism will be part of the application programming language interface to AMOS. We have developed such an interface between AMOS and the functional concurrent programming language Erlang [Arm93] for real-time applications. The callback mechanism is also a part of the communication protocol between AMOS servers.

Possible tasks for the trackers include:

- Notifying the end user that data have changed.
- Refreshing data browsers
- Modifying values in mediators.
- Changing processing heuristics in mediators.
- Changing stored abstractions in mediators.
- Informing applications that data views which the application depends on have changed.

By using active rules the monitor model can filter insignificant data source changes before notifying the application. This decreases the frequency of notification for intensively updated data. Notification filtering is required, for example, by real-time monitoring AI systems where the tracker initiates time-consuming reasoning activities [Was89].

2.3 Domain Models

Domain knowledge and data now hidden in application programs should be extracted from the applications and stored in mediators with domain specific models and operators, called *domain models*. The benefits of using domain models include easier access through a query language, better data description (as schemas), transaction capabilities, and other benefits currently provided only by advanced DBMSs. Examples of domain models are models for structural analysis of mechanical designs, models to obtain a preferred part for a product, or models to describe properties of a user interface. The query processing of AMOSQL must be about as efficient as customized main-memory data structure representations. This would encourage the use of local embedded AMOS databases linked into applications. Domain models often need to be able to represent specialized data structures for the intended class of applications.

Important research problems in developing domain models are to investigate:

1. How is the domain modelled using an OO query and modelling language?
2. Which domain-oriented data structures are required, and how should they be represented?
3. What domain-oriented operators need to be defined?
4. How are queries accessing domain-oriented data structures optimized?

2.4 Locators

In large dynamic information bases, it is not trivial to know which data sources contain requested data. For this, a class of mediators is needed which, given descriptions of the data to retrieve, locates the matching data sources. AMOS mediators, called *locators*, will be developed as servers that know properties of other mediators and where they are located. In a simple environment the application will know exactly where the data sources are located, e.g., by knowing the exact locations of database tables. In a broadly distributed environment one may not have such direct 'handles' to the data sources, but rather query the locators given descriptions of what to look for. Locators provide a query language for connecting data to application programs. The effect is to increase flexibility when information sources are changing.

The need for locator facilities has been acknowledged in the research area of *mobile databases* [Imi92], which combine future telecommunication and database capabilities. In a global and very fast network of information servers, databases are accessible via radio links. When people travel long distances, the system will eventually move data to (or create data in) new locations. In such an environment non-trivial locator facilities become very important. There are some connections between locators and traditional name servers, where IP addresses are looked up via a set of distributed servers. However, since AMOS servers are relatively lightweight, it will be feasible to make each of our locators a complete AMOS server. This will make it possible to provide many new locator services through locator querying. Locators also have connections to traditional DBMS data dictionaries. However, data dictionaries are centralized, i.e. a single data dictionary knows where all data is located, which is what is required to support conventional distributed databases. In contrast, our locators are distributed, autonomous, and loosely coupled.

2.5 Distributed AMOS Systems

The architecture requires facilities to state OO queries and to build OO models that span many AMOS servers. Therefore the system needs to contain means for intercommunication between AMOS servers as well as between AMOS and applications.

We have developed a transactional remote procedure call mechanism that handles low level message interfaces between AMOS servers. A query layer will be built on top of this mechanism. Transactional behaviour ensures that each database can remain consistent after communication or software failures.

We also plan to generalize monitor models so that active rules can be specified that access more than one AMOS server.

3 A Scenario

With AMOS it will be possible to build domain models that combine data from several outside data sources with local data, and which contain rules that assists the user in making decisions.

As an example, consider a computer-supported quotation task, where the suggested design and price depend upon prices from subcontractors, e.g. a HV-transformer design depends on copper and oil prices, or turn-key dairy process equipment depends on stainless steel tubing prices. Integrators allow the information to be stored in and be accessible from different vendor databases.

Product data are represented differently by different suppliers. Integrator models allow conversions between semantically different data representations.

Domain models allow customization of parts of the product selection model by local data and rules. For example, the user might specify preferred price ranges, quality requirements, and constraints on the means for transportation from the supplier. Different domain models will be used by different users and have different customizations.

All data used in the product selection may not be directly available for each considered product and locators must then be used to find the appropriate database. For example, access to each supplier's database is needed in order to estimate the cost of obtaining a product.

Assuming that the main contractor is not in a direct hurry to buy the product, s/he may postpone the purchase until the right market conditions arise which can be provided by monitor models. For example, if supplier A does not have the required product in stock, the main contractor may want a signal if and when it can be delivered from supplier A. Similarly, s/he may want a signal if the price for the product drops below some threshold in the case where a sale is expected. These kinds of monitoring conditions are expressible in the rule language. We also plan to add timeliness specifications in monitoring models, e.g. to specify a deadline after which it is absolutely necessary to have an order put on the needed product, even if the choices are not the best.

4 Summary

The AMOS architecture was described where AMOS information bases mediate between application programs and data sources. AMOS provides facilities to extract data, and to manipulate and model the extracted data using a powerful query and modelling language. The system provides integrator servers that combine data from many different data sources, and provide OO views for all types of data sources. The modelling language has active rule facilities that detect when the state of a data source is updated in some 'critical' way. Critical data source updates can then initiate reasoning in application programs or just notify the user. The modelling language is based on extensions to OSQL [Fis89].

An example scenario was given of the use of AMOS to help main contractors get timely and needed information for the quotation task. One may construct similar scenarios for other domains, e.g., computer network service planning systems, and project planning and tracking systems.

References

[Abi91] Abiteboul S. and Bonner A.:'Objects and Views', in *the Proceedings of ACM SIGMOD*, 1991.

[Ahm91] Ahmed R., DeSmedt P., Du W., Kent W., Ketabchi M.A., Litwin W., Rafii A., and Shan M-C.: 'The Pegasus Heterogeneous Multidatabase System', *IEEE Computer*, Vol. 24, No. 12, Dec. 1991.

[Arm93] Armstrong J., Williams M., and Virding R.*: Concurrent Programming in Erlang*, Prentice-Hall, 1993. ISBN 13-285792-8.

[Bro85] Brownston L., Farell R., Kant E., and Martin A.: *Programming Expert Systems in OPS5*, Addison-Wesley, Reading Mass., 1986.

[Cas93] Castellanos M.: 'Semantic Enrichment of Interoperable Databases', *Proc. RIDE-IMS (Interoperability in Multidatabase Systems) Workshop*, Vienna 1993.

[Cho92] Chomicki J. and Litwin W.: 'Declarative Definition of Object-Oriented Multidatabase Mappings', in Özsu M.T., Dayal U., Vadduriez P. (eds.): *Distributed Object Management*, Morgan Kaufmann Publishers, 1993 (to appear).

[Day89] Dayal U. and McCarthy D., 'The Architecture of an Active Database Management System', in *the Proceedings of ACM SIGMOD*, 1989, pp. 215-224.

[DE92] *IEEE Data Engineering* bulletin, Vol. 15, No. 1-4, Dec. 1992.

[Fis89] Fishman D. et al.: 'Overview of the Iris DBMS', *Object-Oriented Concepts, Databases, and Applications*, ACM press, Addison-Wesley Publ. Comp., 1989.

[Han92] Hanson E. N.: 'Rule Condition Testing and Action Execution in Ariel', in *the Proceedings of ACM SIGMOD*, 1992, pp. 49-58.

[Imi92] Imielinski T. and Badrinath B.R.: 'Querying in highly mobile distributed environments', in *the Proceedings of VLDB '92*, pp. 41-52.

[Lan82] Landers T. and Rosenberg R.: 'An Overview of Multibase', in Schneider H-J. (ed.): *Distributed Databases*, North-Holland, 1982, pp. 153-184.

[Lit92] Litwin W. and Risch T.: 'Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates', *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 6, December 1992.

[Lob93] Loborg P., Risch T., Sköld M., and Törne A.: 'Active Object-Oriented Databases in Control Applications', in *the Proceedings of the 19th Euromicro Conference*, Barcelona 1993 (to appear).

[Lu93] Lu H., Ooi B-C., and Goh C-H.: 'Multidatabase Query Optimization: Issues and Solutions', in *the Proceedings of RIDE-IMS (Interoperability in Multidatabase Systems) Workshop*, Vienna 1993.

[Ris89] Risch T.: 'Monitoring Database Objects', in *the Proceedings of VLDB '89*, Amsterdam 1989.

[Ris92] Risch T. and Sköld M.: 'Active Rules based on Object-Oriented Queries', *IEEE Data Engineering* bulletin, Vol. 15, No. 1-4, Dec. 1992, pp. 27-30.

[She90] Sheth, A. and Larson, J.: 'Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases', *ACM Computing Surveys*, Vol. 22, No. 3, September 1990.

[She91] Sheth, A.: 'Semantic Issues in Multidatabase Systems', Preface in the special issue by editor, *SIGMOD RECORD*, Vol. 20, No. 4, December 1991.

[Shi81] Shipman, D.W.: 'The Functional Data Model and the Data Language DAPLEX', *ACM Transactions on Database Systems*, Vol. 6, No. 1, March 1981.

[Was89] Washington R. and Hayes-Roth B.: 'Input Data Management in Real-Time AI Systems', *11th International Joint Conference on Artificial Intelligence*, 1989, pp. 250-255.

[Wid90] Widom J. and Finkelstein S.J.: 'Set-oriented production rules in relational database system', *Proc. ACM SIGMOD*, Atlantic City, New Jersey, 1990, pp. 259-270.

[Wie92] Wiederhold G.: 'Mediators in the Architecture of Future Information Systems', *IEEE Computer*, March 1992.

15.4 Paper IV

P. Loborg, T. Risch, M. Sköld, and A. Törne: Active Object-Oriented Databases in Control Applications, in Proceedings of the 19th Euromicro Conference, Barcelona, September 1993.

Active Object-Oriented Databases in Control Applications

Peter Loborg, Tore Risch, Martin Sköld, Anders Törne

Dept. of Computer and Information Science, Linköping University
S-581 83 Linköping, Sweden
E-mail: petlo, torri, marsk, andto@ida.liu.se

Abstract

This paper describes a unified architecture for control applications using an extended object-oriented database system with queries and rules. We specify the requirements that control applications demand on the database, and how they are met by our database system architecture. The database system, AMOS[1], is a main memory database that provides information sharing, powerful data access via an object-oriented query language (AMOSQL), data independence, and reactive behavior by active rules. The application considered is a robot and manufacturing instruction system, ARAMIS, which is a task-level programming system. The presentation describes a specific scenario related to manufacturing control.

1. INTRODUCTION

Control applications are a significant and important part of the industrial use of information technology. Control may be described as the means to control or restrict the behaviour of external world processes, so that a “correct” or “intended” behaviour is achieved. This might be done by software, specially designed hardware, human intervention or mechanical devices. We will here discuss design and architectural issues with regard to control by software.

Typical for the software design problem in control applications are the requirements originating from timing constraints, some derived from pure control considerations and others from the external processes themselves. These requirements put constraints on the software execution times [7][19]. Much research effort has been put into methods and formalisms for timing analysis to guarantee the timing requirements of the system - so called hard real-time systems.

This paper will not focus on this issue, but rather on the *data management* problems which arise when the controlled and controlling systems become large, composite, and complex, and when the controlling subsystem involves human operators.

1. This work is partially supported by the Swedish National Board for Industrial and Technical Development (NUTEK) and by CENIIT, Linköping University.

Traditionally the understanding of control by software focuses on fully automatic control at algorithmic level or at a level close to the external process [23]. At this level the data management problems are small. Usually there is no problem with sharing data between different applications or use of data, since data is local and normally not used outside the local algorithm or control loop. However, as soon as the controlled system becomes larger and more complex, typical data management and information processing problems arise.

The contemporary most popular approach to handling the design of complex software systems is object orientation. A common approach when applying this to control by software is to use object-oriented (OO) modelling and programming to structure the software [1][13]. This structure usually reflects the physical structure of the controlled system. Although this aspect of the design is important, the approach ignores the problems with loosely coupled control processes possibly involving several human operators, e.g., nuclear power plants or large manufacturing facilities, which have a high degree of autonomy and parallelism. This extension of the problem domain makes the management problem of the shared data obvious. The present paper focuses on this problem.

One of the important issues is the separation of shared data and local data for control. All the different control processes share the same data model. However, data independence will be achieved by allowing each process to have its own views on shared data. The shared data model is called a *world model* (WM). These properties are achieved by representing the WM in an OO database system having query and logical data view capabilities.

The use of databases in real-time control systems has recently attained increasing interest. Some of the relevant issues are discussed in [7][9][14][16][17].

We are developing a robotics and manufacturing instruction and runtime system, ARAMIS, for manufacturing applications, and a next generation database architecture, AMOS [4], as a general framework for engineering databases. AMOS has an OO Query Language AMOSQL, which is an extension of OSQL [5]. AMOSQL has transactions, active rules, and extensibility by foreign functions. This paper will discuss a unified approach for control applications, where the world model is represented in an OO database with query and rule capabilities, like AMOS, with ARAMIS as the control application environment.

The next section will present a scenario to be used in the later sections. Section 3 and 4 will present the ARAMIS world model and the AMOS architecture. Section 5 will discuss the unified database architecture for control applications and then we conclude with a general discussion.

2. AN EXAMPLE SCENARIO

In following sections a scenario from a manufacturing application will be used to exemplify the use of active OO databases in control applications.

The task in the scenario is to assemble a subassembly of, for example, an airplane. The resources available are a manipulator and a fixture to perform the

assembly itself. Some of the parts needed in the subassembly arrive via a conveyor belt in the order necessary to perform the assembly. These parts originate from part feeders. Finished subassemblies are placed on a pallet.

The functional requirements on the application is that the assembly should only start if all needed parts are available to the manipulator during the assembly process. If feeder storage is low, it should be filled, manually or by automatic guided vehicles, from central storage. Some suboperations of the different processes need preconditions to be fulfilled before starting, e.g., the PICK-UP operation of the manipulator has as a precondition that the part is available in the pick-up position and the conveyor belt is secured (locked).

The task can naturally be decomposed into processes - the assembly process itself involving the manipulator and the fixture, the part transport process for feeding parts onto the conveyor belt and positioning them for the pick-up operation, and the central storage fetching process which serves this assembly cell together with many others.

3. THE ARAMIS WORLD MODEL

The ARAMIS system is an instruction and runtime system for control of manufacturing environments. The system has been designed with a layered architecture based on different levels of abstraction [21].

3.1 Description

ARAMIS consists of three different programming levels, *the task level*, *the control level* and *the physical level*. The task level specifies what operations should be performed in the physical environment and under what conditions. It uses a graphical rule-based hybrid language with primitives for creating parallel execution threads dynamically and synchronization of different parallel threads [12]. The concept corresponding to a task is called a *worker*, which can contain parallel threads by using the parallelisation primitives. The task level program (a set of workers) executes by setting reference values for the objects in a world model (WM). The WM is shared between the different tasks (workers) executing in the physical environment. Basically this corresponds to a blackboard model for communication [8], although we emphasize the database aspects. The model data may be functions or sensor values or they might be implicitly known by the known postconditions resulting from executed actions. One of the benefits of the model is that no difference is made between these forms of knowledge about the world state.

The control level is responsible for keeping the real world in a state represented by the world model. This level therefore mimics a *servo mechanism* for the whole system. The programming at this level is typically done by control engineers in a traditional language.

The physical level is the actual connection to the real world, where explicit I/O is performed with sensors and actuators.

One of the important aspects of the world model is that every variable has

two values, the *set value* and the *get value*. Set values have the semantics of reference values for the servo mechanism and the get values have the semantics of “known” state parameters - actual values.

3.2 The execution model

Each object in the world model (also called an *active component*) is represented as a deterministic finite automaton (DFA), augmented with control algorithms to be executed for each state transition and state [11]. Each DFA may be in one of two states - transiting between states or being held in a state by the control level. Concurrent transitions between objects are orchestrated by the task level.

The execution model implies that state transitions have a time duration. During the state transition the value of the changing model values are considered as unknown, unless explicit reference to a sensing model is made.

The graphical language for describing tasks is, as was indicated above, able to represent conditional sequencing and possible parallelism of the subactions in each task (worker).

The tasks (workers) model reactive behaviour.

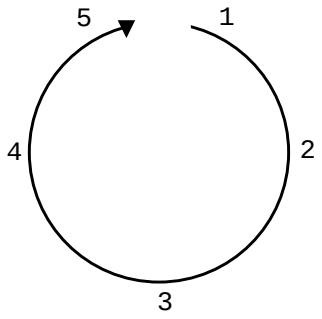


Figure 1. The ARAMIS execution cycle

1. external event
2. possible satisfaction of conditions in the world model
3. triggering the execution of one or several tasks
4. execution of possibly parallel subactions
5. requests for possibly concurrent state transitions in the WM objects

3.3 The scenario data model

In this paper a data model of the scenario in section 2 will be given as an example. It is possible to create the complete corresponding database model and

worker specifications from this description (if completed).

The manipulator, the conveyor belt, the fixture, the feeders and the pallets are represented as active components. Furthermore the parts are active components because they have state and restrictions on their state transitions. However, they only have a passive role for the control process as information carriers in the world model, and are therefore not needed for the following presentation.

Objecttype	**Attribute/ Function**	**Valuetype**
Workcell	assembles	Subassembly
	transport_of	Transport
	manipulator_of	Manipulator
	feeder_of	Feeder
	out_pallet_of	Pallet
Manipulator	state_of	{OK, idle, busy}
Transport	at_pickup_location	Part
	state_of	{locked, moving}
Feeder	state_of	{OK, empty, needs-refill}
	feeds_part	Part
Pallet	state_of	{OK, empty, full}

The objects will be instantiated for different configurations of workcells.

3.4 Requirements on the database realization

The ARAMIS model presented briefly above have certain requirements on a database realization:

- Performance is obviously important, since control actions and reactivity should be as timely as possible. Database functionality always imposes overhead performance costs, but they should be minimized. Any control action or reactive action with timing requirements faster than is realizable in the database must be modelled at the control level, thereby losing the benefits of database management.
- The response time of the database must be predictable, i.e., it is not acceptable that the same database operation performs significantly different from time to time. This is not the case if, for example, data access is dependent on whether sought data are available in a buffer or not, as in the case of disk-based DBMS.
- Object-oriented modelling and access via a query and data definition language is desirable. This levels the database approach for control applications with OO programming approaches. The query language gives declarative data access functionality.
- Extensibility is required to execute actions and to perform sensing.
- Heterogeneous database access is important when control actions are dependent upon data of traditional character, like in-stock figures or exchange rates.

- The reactive behaviour requirements in control applications demand active database behaviour.
- Transactions and other error recovery mechanisms must be supported.

4. THE AMOS ARCHITECTURE

The AMOS (Active Mediators Object System) architecture [4] uses the *mediator* approach [22], which introduces an intermediate level of software between the database and its use in applications and by users. We call our class of intermediate modules *active mediators*, since they support 'active' database facilities.

The AMOS architecture is built around a *main memory* based platform for intercommunicating object-oriented databases. Each AMOS server has DBMS facilities, such as a local database, a data dictionary, a query processor, transaction processing, remote access to data sources, etc. Main-memory database processing is necessary for control applications in order to achieve fast and predictable response times. In AMOS the disk is used for background back-up purposes only.

4.1 Object-Oriented Queries

A central component of AMOS is an object-oriented (OO) query language AMOSQL that generalizes OSQL [5]. AMOSQL supports OO abstractions and declarative queries, which makes it possible to declaratively specify different object views for different applications.

The system is extensible through *foreign functions* written in an external programming language (usually Lisp or C), e.g., to access sensor data [16], or to start actuator action.

To support the initial work on AMOS, a main-memory OO DBMS engine is used [10]. A query optimizer translates AMOSQL queries and methods into optimized execution plans in an internal logical language, ObjectLog.

4.2 Active rules based on OO queries

AMOS supports 'active rules' as an extension of OSQL [15]. In an active rule a procedure is executed when the database reaches a specified state. The rules are of the type:

```
when query(parameters)
do exec procedure(parameters)
```

The query in the rule condition can be any AMOSQL query and specifies when the rule should be triggered. The action part can be any AMOSQL database operation.

These types of rules are more powerful than ordinary database triggers or 'ECA' rules [2], since the entire condition for the triggering of a rule is specified through a declarative query. Rules can be parameterized and overloaded on different types in the database. The execution of rules is made efficient by

using incremental computation of rule conditions and by using efficient optimization techniques of the involved queries.

4.3 Transactions

AMOS supports atomic transactions so that database updates are rolled back in case an error occurs. The 'rollback routines' can be programmed to customized clean-up, e.g., to restore the external world to the new state after a rollback of the world model database.

4.4 Heterogeneous database access

A distributed AMOS architecture is being developed where several AMOS servers communicate, and where queries in a multi-database language are allowed to refer to other autonomous AMOS servers, relational databases, sensors, and other data sources [4]. A relational database system, SYBASE, is currently being integrated with AMOS. A particular problem with such an integration is to get OO access to non-OO data sources. The method allows OO queries to be stated with transparent access to non-OO data sources. It will be possible to state queries that combine sensor data with, for instance, conventional databases.

5. A UNIFIED ARCHTECTURE FOR CONTROL APPLICATIONS

5.1 Declarative modelling/access via OSQL

The representation of the WM of ARAMIS in a database provides powerful data access through the query language. The object-oriented data modelling language of AMOS, AMOSQL, has the benefits of a traditional OO language by providing a type system with an inheritance mechanism over subtypes. Furthermore, by accessing the database through object views defined by a query language, data independence between the database and the rest of the system can be achieved, since a data access query can be made in the same format, even after the database structure has changed.

Access to sensors can be implemented as external side-effect free function calls from AMOSQL [10]. This would allow the application programs and/or the operator to state arbitrary complex queries over the current state of the world model - superior to ad hoc navigational database access.

5.2 Active rules in control applications

Active rules in the database can be used for two basic functions in the ARAMIS architecture.

The starting conditions for ARAMIS processes (workers) can be compiled to AMOSQL rules. Starting conditions in AMOS are conditions over the WM, and are thus monitored more efficiently, directly in the database. A starting condition compiled as an AMOSQL rule awakes the associated process when

the starting condition becomes true.

The servo mechanism in the ARAMIS architecture can be implemented as active rules ranging over properties of active components in the WM. The rules can be defined for specific component instances or for whole component classes. The servo mechanism can be implemented by an interplay between ARAMIS actions, AMOSQL rules, and control algorithms.

The rules have conditions that are sensitive to state changes of particular active components and actions that call algorithms in the control system or awakes the ARAMIS inference machine.

The servo mechanism will consist of three phases:

- An ARAMIS process changes the state of the WM and is suspended.
- An AMOSQL rule detects the change and calls a control algorithm.
- The control algorithm executes and changes the physical state of the controlled system to match the state in the WM. Upon completion the rule calls the ARAMIS inference machine to awaken the suspended process.

5.3 Heterogeneous database access in control applications

The distributed and heterogeneous database access capabilities provided by AMOS have the following benefits for ARAMIS:

- Uniform access to heterogeneous data makes it easy to extend the WM with access to conventional databases. It will be possible to state AMOSQL queries combining control and conventional data. For example, error messages can refer to manufacturing data for robot parts, which are accessible from a relational database. Similarly, activity reports can be printed that combine control and relational data.
- Data distribution will make it possible to have geographically distributed ARAMIS systems, each having its own WM views, but also sharing parts of the WM with other ARAMIS processes.
- By generalizing active rules to be distributed over several databases, one may coordinate the behavior of several ARAMIS processes so that WM updates of one process remotely triggers actions in other processes.

5.4 Error detection and recovery

As the ARAMIS system is divided into a task level and a control level, so is the error handling. Furthermore, the error handling (at each level) can be viewed as twofold: handling anticipated errors and unanticipated ones. Different techniques may be used to handle each case.

At the control level each request for a state change (a state transition) is viewed as a transaction. However, using pure transactions as a base for error recovery at this level is not always possible, since there are irreversible state transitions and transitions where the inverse transition is composed of several transitions through some intermediate states. Classifying the transitions as ‘continuable’, ‘undoable’, etc., and augmenting them with extra information is one possible approach to handle ‘anticipated’ errors [20]. Therefore, the state transition should be modelled as a collection of coupled transactions, e.g.,

SAGAs [6] or activity models [3]. Some of the transitions do not guarantee that the resulting state is consistent, i.e., there might be transitions that may abort and report an error, as well as those that fail (but leave everything in a consistent state) and report the failure. The reasons leading to the abortion or failure of a transition may be internal (programming errors) as well as external - in the latter case, either detected by the algorithm itself, by operator intervention or by active rules monitoring the state transition (e.g., prevail condition checks).

At the task level active rules can be used as exception handlers, i.e., to handle more or less anticipated errors. For unanticipated errors a combination of manual intervention and planning is required.

```
create function assembles(Workcell) -> Subassembly;

create function feeder_of(Workcell) -> Feeder;

create function out_pallet_of(Workcell) -> Pallet;

create function parts_of_subassembly(Subassembly)
    -> bag of Part;

create function state_of(Feeder) -> Charstring;

create function state_of(Pallet) -> Charstring;

create function feeds_part(Feeder) -> Part;

create function ready_parts(Workcell c) -> Part p as
    select p for each Feeder f
    where p = feeds_part(f) and
    f = feeder_of(c) and
    state_of(f) != “empty”;

create rule ready_to_go(Workcell c, Worker w) as
    when in(parts_of_subassembly(assembles(c)),
            ready_parts(c)) and
         state_of(out_pallet_of(c)) != “full”
    do activate_worker(w); /* procedure that calls
                              ARAMIS */
```

Figure 2. A worker initiation condition modelled by an AMOSQL rule

5.5 The Scenario

The scenario in section 2 can be implemented as a number of AMOSQL functions and rules.

In figure 2 the condition for activation of an assembly worker is monitored by the rule `ready_to_go`. The function `parts_of_subassembly` returns the parts of a particular subassembly. The function `ready_parts` returns all the parts that are ready to be feed onto to the transporter. The condition in the rules checks that all the parts needed for the subassembly are present in the feeders. In figure 3 the condition for the PICK-UP operation is monitored by the rule `ready_to_pickup`. The condition in the rule checks that an object is in the pick-up location on the transporter, that the transporter is locked and that the manipulator is not busy.

```
create function manipulator_of(Workcell) -> Manipulator;

create function transport_of(Workcell) -> Transport;

create function at_pickup_location(Transport) -> Part;

create function state_of(Transport) -> Charstring;

create function state_of(Manipulator) -> Charstring;

create rule ready_to_pickup(Manipulator m) as
    when for each Workcell c, Transport t, Part p where
    state_of(m) != “busy” and
    m = manipulator_of(c) /* find c given m */ and
    t = transport_of(c) and
    p = at_pickup_location(t) and
    state_of(t) = “locked”
    do pickup(m, p); /* activates a control algorithm */
```

Figure 3. An operation invocation condition modelled by an AMOSQL rule

6. DISCUSSION

Our approach has the following advantages vis-a-vis conventional approaches.

The world model may be designed at a very high level using OO abstractions and declarative queries.

The world model is easy to access using OO queries. Sensor data can easily be made accessible from within AMOSQL queries.

Incremental modification of the world model is supported by, for example, adding new functions, rules, data sources, actuators, etc. Much flexibility is gained.

The transaction management of AMOS can be utilized to guarantee atomic updates of the world model, even when much data is updated simultaneously and concurrently. This guarantees world model data consistency after more or less complex updates.

The main-memory representation of the database, guarantees predictable and fast response times.

REFERENCES

[1] T. E. Bihari and P. Gopinath: Object-Oriented Real-Time Systems: Concepts and Examples, *IEEE Computer*, **25**, 12, 25-32, 1992.

[2] U. Dayal and D. McCarthy: The Architecture of an Active Database Management System, in t*he Proceedings of ACM SIGMOD Conference*, 1989, pp. 215-224.

[3] U. Dayal, M. Hsu, and R. Ladin: Organizing Long Running Activities with Trigger and Transactions, in t*he Proceedings of ACM SIGMOD*, May 23-25, Atlanta City, 1990, pp. 204-214.

[4] G. Fahl, T. Risch, and M. Sköld: AMOS - An Architecture for Active Mediators, *NGITS'93*, Haifa, Israel, 1993 (to be published).

[5] D. Fishman, et. al.: Overview of the Iris DBMS, *Object-Oriented Concepts, Databases, and Applications*, ACM press, Addison-Wesley Publ. Comp., 1989.

[6] H. Garcia-Molina and K.Salem: Sagas, *Proc. SIGMOD*, May 27-29, 1987, San Francisco, pp. 249-259.

[7] M. H. Graham: Issues in Real-Time Data Management, *J. Real-Time Systems*, **4**, 185-202, 1992.

[8] B. Hayes-Roth: A blackboard architecture for control, *Artificial Intelligence*, Vol. 26, pp. 251-321, 1985.

[9] J. Huang: Extending Interoperability into the Real-Time Domain, *Research Issues in Data Engineering: Interoperability in Multidatabase Systems, RIDE-IMS'93*, Vienna, Austria, IEEE Computer Society Press, April 1993.

[10] W. Litwin and T. Risch.: Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates, *IEEE Transactions on Knowledge and Data Engineering*, **4**, 6, Dec. 1992.

[11] P. Loborg, M. Sköld, A. Törne, and P. Holmbom: A Model for the Execution of Task Level Specifications for Intelligent and Flexible Manufacturing Systems, in *Proceedings of the Vth Int. Symposium on Artificial Intelligence, ISAI92*, Cancun, Mexico, Dec. 1992.

[12] P. Loborg and A. Törne: A Hybrid Language for the Control of Multimachine Environments, in *Proceedings of EIA/AIE-91*, Hawaii, June 1991.

[13] O. Z. Maimon and E. L. Fisher: An Object-Based representation Method for a Manufacturing Cell Controller, *Artificial Intelligence in Engineering*, **3**, 1, 2-

11, 1988.

[14] K. Ramamritham: Real-Time Databases, *Distributed and Parallel Databases*, **1**, 2, April 1993.

[15] T. Risch and M. Sköld: Active Rules based on Object-Oriented Queries, *IEEE Data Engineering* (Quarterly), Jan. 1993.

[16] R. Snodgrass: A Relational Approach to Monitoring Complex Systems, *ACM Transactions on Computer Systems*, **6**,2, May 1988, pp. 157-196.

[17] S. H. Son: Real Time Database Systems: A New Challenge, *IEEE Data Engineering*, **13**, 4, 51-57, 1990.

[18] J. A. Stankovic, and K. Ramamritham: *Hard Real-Time System*s, Tutorial, IEEE, 1988.

[19] J. A. Stankovic: Misconceptions about Real-time Computing, *Computer*, **21**, 10, 10-19, 1988.

[20] U. Schmidt: A Framework for Automated Error Recovery in FMS, in t*he Proceedings of 2nd International Conference on Automation, Robotics and Computer Vision*, Singapore, 1992.

[21] A.Törne: The Instruction and Control of Multi-Machine Environments, in t*he Proceedings of Applications of Artificial Intelligence in Engineering V,* Springer-Verlag, vol. 2, Boston, July 1990.

[22] G. Wiederhold: Mediators in the Architecture of Future Information Systems, *IEEE Computer*, March 1992.

[23] K. J. Åström and B. Wittenmark: *Computer Controlled Systems*, Prentice Hall, N.J., 1984.

15.5 Paper V

M. Sköld, E. Falkenroth, and T. Risch: Rule Contexts in Active Databases - A Mechanism for Dynamic Rule Grouping. in the Second International Workshop on Rules in Database Systems (RIDS'95), Athens, Greece, September 25-27, 1995, Springer Lecture Notes in Computer Science, ISBN 3-540-60365-4, Pages 119-130, 1995.

Rule Contexts in Active Databases
- A Mechanism for Dynamic Rule Grouping

Martin Sköld, Esa Falkenroth, Tore Risch
Department of Computer and Information Science, Linköping University
S-581 83 Linköping, Sweden
e-mail: {marsk,esafa,torri}@ida.liu.se

Abstract. Engineering applications that use Active DBMSs (ADBMSs) often need to group activities into modes that are shifted during the execution of different tasks. This paper presents a mechanism for grouping rules into *contexts* that can be activated and deactivated dynamically. The ADBMS monitors only those events that affect rules of activated contexts.
By *dynamic* rule grouping the set of monitored rules can be changed during the transactions. In a *static* rule grouping the rules are associated with specific objects during the schema definition.
A rule is always activated into a previously defined context. The same rule can be activated with different parameters and into several different contexts. Rules in a context are not enabled for triggering until the context is activated. However, rules can be directly activated into a previously activated context. When rule contexts are deactivated all the rules in that context are disabled from triggering.
The user defined contexts can be checked at any time in a transaction. Rule contexts can be used as a representation of coupling modes, where the ADBMS has built-in contexts for immediate, deferred, and detached rule processing. These built-in coupling modes are always active and are automatically checked by the ADBMS.
Contexts and rules are first-class objects in the ADBMS. Database procedures can be defined that dynamically activate and deactivate contexts and rules to support dynamically changing behaviours of complex applications.
The context mechanism has been implemented in the AMOS ADBMS. The paper concludes with an example of a manufacturing control application that highlights the need for rule contexts.

1 Introduction

A system for building manufacturing control applications was implemented using an ADBMS [10]. In the system active rules control the manufacturing tasks. Details about the system and examples of active rules are presented in section 4. Results from this system integration are:

- These type of engineering applications need to group activities into modes that are shifting during the execution of different tasks.

- Since the ADBMS did not initially have mechanisms for handling mode changes the application had to implement this functionality by introducing state variables in the rule conditions.

- The state variables caused the rules to become complex and unintuitive. A rule would often need to refer to several different state variables.

- The state variables represent control knowledge. It is better to separate rules representing domain knowledge from rules for control knowledge, e.g. by defining *meta-rules* [1][2].

- Implementing mode changes by altering state variables is inefficient since the total number of simultaneously monitored rules will be unnecessarily large. By having the ADBMS support mode changes internally the overhead for rule checking can be kept low.

This paper presents an ADBMS mechanism for dynamically grouping rules into *contexts* that are activated and deactivated dynamically. The contexts are associated with different modes in the applications. When the application shifts between modes, the ADBMS is ordered to shift attention, or *focus*, to the associated rule context. Shifting between contexts means that all rules in the old context are ignored and the rules in the new context are monitored instead. There are applications that need to work with modes on different levels where a mode can consist of many hierarchically ordered sub-modes. This means that the ADBMS must be able to handle several rule contexts simultaneously and to support modelling of contexts in the schema. By defining contexts and rules as first-class objects in the ADBMS this is accomplished. This approach also supports the definition of *meta-rules* that are defined over rules and contexts.

Applications must not only have the possibility to create and delete contexts and activate and deactivate them, but must also be able to control when the rules are to be checked. For example, the application might initiate a series of operations and then check if any rules were triggered. This usually falls outside of the general coupling modes defined in ADBMSs. Our contexts therefore have *rule processing points*, which allow applications to define their own coupling modes where the rules can be checked at a user-specified time in a transaction.

The contexts are also used internally in the ADBMS to implement system coupling modes. System coupling modes are associated with predefined contexts that are automatically checked by the system.

The paper presents rules and rule contexts as implemented in the AMOS (Active Mediating Object System)[5][13] ADBMS. The paper concludes with an example of a manufacturing control application that highlights the need for rule contexts.

2 Related Work

The idea of grouping rules dynamically into different contexts was initially developed in expert systems [1][17]. Other names for these groups of rules include *worlds* and *viewpoints*. In expert systems these rule contexts are usually used for organizing different *hypotheses* during a deduction process. In an ADBMS the issue is more of organizing the different *activities*.

The contexts were also supported in the rule-based expert system Mycin and its successor Oncosin [1]. In Mycin contexts had to be specified as special context varia-

bles in rule conditions; in Oncosin a special CONTEXT clause on each rule referred to the context variables. By contrast our contexts completely separate the context specifications (i.e. the control information) from the rules (i.e. the knowledge) and therefore the same rule can occur in many contexts with different control strategies.

The rules in an ADBMS are often defined as first-class objects in the database schemas [3]. In Object-Oriented (OO) systems the rules can often be grouped as belonging to a class and rules can be associated with other classes in a similar way to class methods. KEE [8] used this model for grouping rules into *worlds*. This classification is useful when associating rules with specific objects statically, e.g. when associating some constraint on the possible values of a class attribute or reacting to a user-defined event associated with an object. These kinds of rules are usually always active and are triggered when a method is invoked of an instance of the class. However, in many applications there is a need to dynamically group rules that are associated with many different classes of objects.

Both POSTGRES[15] and Starburst[18] allow rules to be members of *rule sets*, which can be ordered hierarchically and where complete rule sets can be activated and deactivated. Rule sets are checked at certain *rule processing points*. The contexts in AMOS are more dynamic since the same rule can be activated in different contexts for different parameters, i.e. for different object instances. The contexts are objects and thus can be stored in any data structure and can be used for relating different data to different contexts. In AMOS contexts are also used for defining built-in coupling modes for rule execution. This means that these contexts have rule processing points that are automatically executed by the system. Since the same rule can be activated into different contexts the same rule can also be given many coupling modes.

In [16] a model is presented for defining applications in terms of *brokers* that represent reactive system components and *roles* that specify the responsibilities of brokers in various situational and organizational contexts. A proposal was made to implement rules using rules and special role-dependent state variables. As was mentioned earlier, we believe this kind of modelling is better supported by rule contexts in the ADBMS.

We define rule contexts as first-class objects to enable functions to be parameterized with contexts, organizing them hierarchically in data structures, and defining rules that manage (create/delete, activate/deactivate) other contexts than their own. This makes it possible to define *meta-rules* as in [1][2] where the meta data consists of other rules and contexts.

3 Rules and Contexts in the AMOS Active DBMS

Active rules have been introduced into AMOS[5][13], an Object Relational DBMS. The data model of AMOS is based on the functional data model of Daplex[14] and Iris[6]. AMOSQL, the query language of AMOS, is a derivative of OSQL[11]. The data model of AMOS is based on objects, types, functions, and rules. Everything in the data model is an object, including types, functions, and rules [3]. All objects are classified as belonging to one or several types, i.e. classes. Functions can be stored, derived, or foreign. Stored functions correspond to object attributes in an OO system and to base tables in a relational system, derived functions correspond to methods

and relational views, and foreign functions are functions written in some procedural language[1]. Database procedures are defined as functions that have side-effects. AMOSQL extends OSQL[11] with active rules, a richer type system, and multidatabase functionality.

3.1 Contexts

When rules are activated in AMOS, they are always associated with rule contexts. The contexts are first-class objects and are created by the statement:

create context *context-name*

where the *context-name* is a global name. Contexts are deleted by:

delete context *context-name*

The contexts are initially *inactive* which means that before a context is *activated* the events affecting its rules are not monitored (unless the events are monitored by another already active context). Contexts are activated by:

activate context *context-name*

which enables all the activated rules in the context to be monitored. Contexts are deactivated by:

deactivate context *context-name*

which disables all the activated rules in the context from being monitored. Two built-in contexts, named `deferred` and `detached`, are predefined and always active for deferred and detached rules, respectively. These are checked automatically by the system. Deferred rules are checked immediately before transaction-commit and detached immediately after.

3.2 Rules

AMOSQL supports Condition Action (CA) rules. The condition is defined as an AMOSQL query and the action as an AMOSQL procedural expression.
The syntax for rules is as follows:

create rule *rule-name parameter-specification* **as**
    when *for-each-clause* | *predicate-expression*
    do *procedure-expression*
where
for-each-clause ::=
    for each *variable-declaration-commalist* **where** *predicate-expression*

The *predicate-expression* can contain any boolean expression, including conjunction, disjunction, and negation. The rules are deleted by:

delete rule *rule-name*

1. In AMOS foreign functions can be written in Lisp or C.

The rules are activated and deactivated separately for different parameters. Rules are activated in different contexts, where the default context is the `deferred` context:

activate rule *rule-name parameter-list* [**strict**] [**priority** 0|1|2|3|4|5] [**into** *context-name*]

deactivate rule *rule-name parameter-list* [**from** *context-name*]

The semantics of a rule in an active context 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. This ensures that we only react to net changes, i.e. *logical events*. A non-empty result of the query of the condition is regarded as *true* and an empty result is regarded as *false*. Since events are not monitored in inactive contexts, rules in them will not trigger until the context is activated and some event happens that causes the condition to become true. *Strict* rule processing semantics guarantees that a rule is triggered only once for each change that causes its condition to become true. Rule priorities can be used for defining conflict resolution between rules that are triggered simultaneously in the same context.

3.3 Rule Contexts and Rule Processing Points

Each context in AMOS has a separate *rule processing point* where the conditions of the rules in the context are checked and where the corresponding actions are executed if the condition is true. (For strict semantics the action is executed only if the condition was false in the previous processing point of the context).

A processing point is either *explicit* or *implicit*. Explicit processing points are issued by explicit calls from applications to a *check* system procedure. Implicit processing points are issued by the ADBMS at specific database states, e.g. just before (deferred rule processing) and after (detached rule processing) each commit point.

Rule contexts can be used as a representation of coupling modes [4]. The coupling modes are defined as named contexts with implicit processing points. All rules that are activated in the same context also have the same coupling mode, i.e. the same rule processing point. Traditionally coupling modes have been associated directly with individual rules. By associating the coupling modes with rule contexts a more flexible model can be achieved. Since rules can be activated into several contexts they can also be given several coupling modes. Coupling modes for *immediate*, *deferred*, and *detached* rule processing can be defined as built-in contexts that are automatically checked by the transaction mechanism of the ADBMS (fig. 1). In [4] a separation was made between E-C and C-A coupling modes. When we refer to immediate coupling mode here, we really mean immediate-immediate, and by deferred we mean deferred-deferred. Contexts for other E-C and C-A combinations could also be defined. Immediate rule checking is currently not supported in AMOS, but its processing points would have to be just after (or before) triggering database operations. User defined contexts with explicit processing points can be checked at any time within a transaction. The detached coupling mode is important in a multida-

tabase architecture like AMOS. In such an architecture one agent or broker may need to monitor the behaviour of another agent [12]. This monitoring must be made on committed data. By using a detached coupling mode the rules that perform the monitoring will never trigger on uncommitted changes.

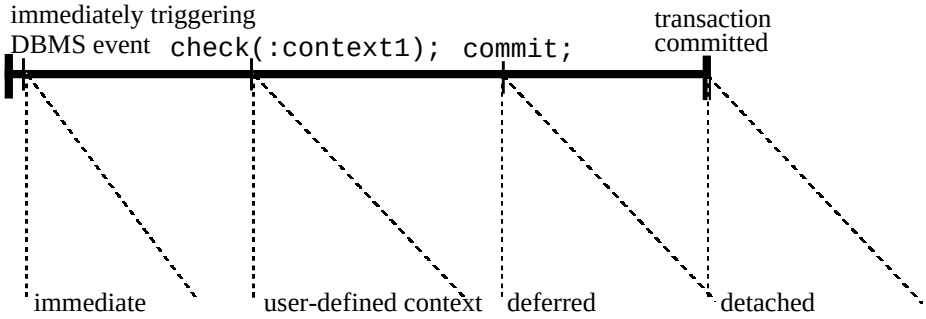


Fig. 1. Contexts as a representation of coupling modes

Decoupled and *causally dependent decoupled* coupling modes [4] can be implemented using general transaction mechanisms for creating sub-transactions and synchronizing transactions.

4 A Manufacturing Control System

The need for a context mechanism became apparent when an ADBMS was used in the implementation of a system for building manufacturing control applications [10]. ARAMIS (A Robot And Manufacturing Instruction System) [9] is a high-level language and a set of tools for designing intelligent behaviour of control systems. The ARAMIS language has similarities with workflow languages [7], but is oriented towards specifying the high-level activities of control applications. The low-level control programs that interact with the physical hardware are isolated from the application programmer by the World Model (WM) metaphor. All the objects in the model of the manufacturing task can be observed and manipulated as objects in the WM. The original ARAMIS system [9] was fully implemented (controlling a robot with various sensors), but with a primitive ADBMS. In [10] a three-level architecture combining the ARAMIS language and an ADBMS is presented. In CAMOS (Control Applications Mediating Object System), see fig. 2, a manufacturing task is expressed in a high-level task language that is partly compiled into an AMOS database that

stores the WM and monitors changes to the objects in the WM.

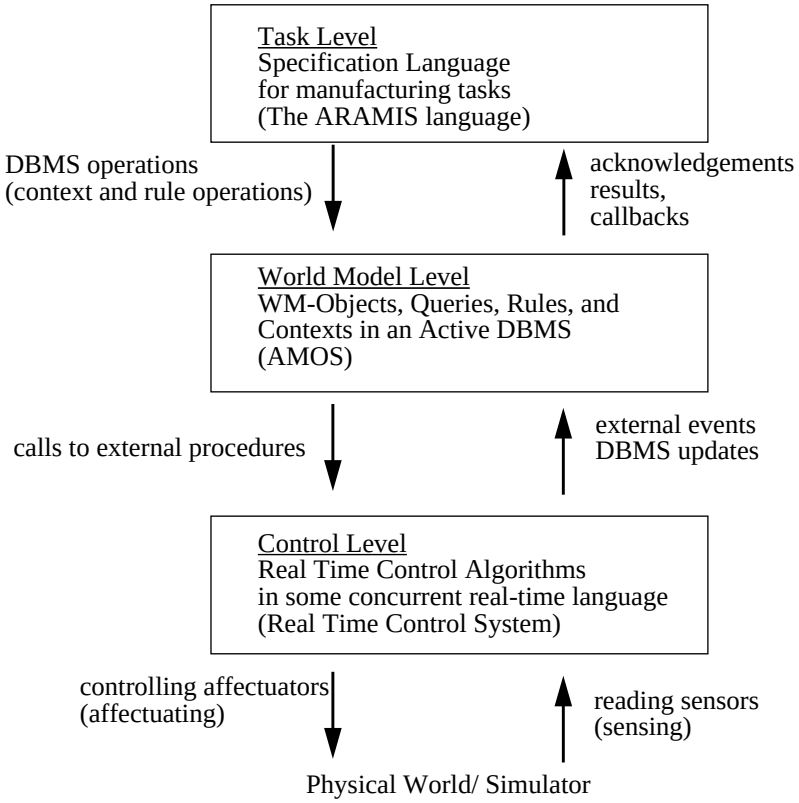


Fig. 2. The three-level architecture of CAMOS

The WM is synchronized with a *physical world* or a *simulator* by cooperation between a *control system* and an ADBMS through a servo mechanism. When the task level updates the WM, the control level affects the physical world to correspond to the WM. Likewise, when the control level senses changes in the physical world, it updates the WM. In the CAMOS architecture the high-level query language and active rules of AMOS are used to support much of the functionality in the WM, e.g. to monitor changes to the WM. Parts of this architecture have been implemented to verify the ideas. Instead of using actual hardware, a simulator of a production cell was used[1]. In the initial implementation state variables were used to model mode changes. Below follows an example of how rule contexts in AMOS can be used instead.

4.1 A Production Cell Simulation

A production cell consisting of a two-armed robot, an elevating rotary table, a press,

1. Based on a simulator developed by Artur Brauer at University of Karlsruhe.

a crane, and two conveyor-belts produces body parts for cars (fig. 3). Unprocessed parts arrive from the left on the lower conveyor-belt and are transported to the elevating rotary table that puts them into gripping position for the first arm (`:arm1`) of the robot. The robot moves a part to the press that presses it into a finished body part. The robot then moves the part, using the second arm (`:arm2`), to the top conveyor-belt that moves to the left. A crane finally picks up the parts and place them on a pallet (lower left of fig. 3).

This is an application that requires a database for storage of data relating to the different parts in stock and also active database management for the actual control of the production task. Another requirement is that the setup should be flexible and the production cell should easily be reconfigured for production of different parts.

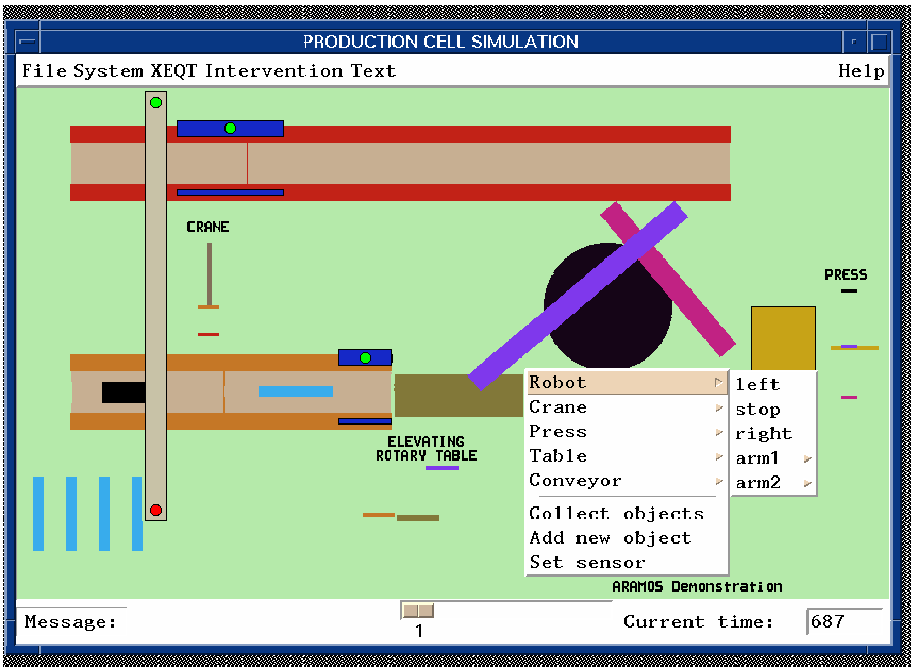


Fig. 3. A top-view of a simulated production cell for manufacturing car body parts

Take a scenario where the production cell can alternate between the production of two different parts. This can be modelled by two different contexts (fig. 4). Each context is used to relate to data needed for each part. Rules that are specific for each different part are activated into the respective contexts. Sub-contexts can also be defined for different activities within the cell. This is illustrated here by two contexts used in both production tasks, one for rules relating to the elevating rotary table and one for the press. There will of course be more contexts and rules, but these are enough to illustrate the idea.

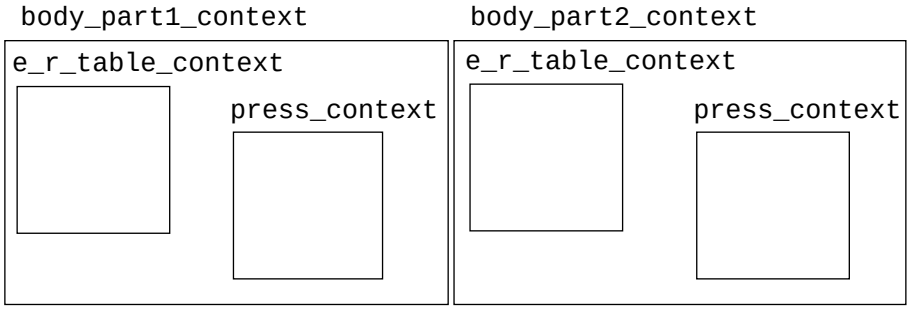


Fig. 4. Example contexts for producing two different parts and two general sub-contexts

An example of a task program for producing `part1` is shown in fig. 5. It is a cyclic program that keeps producing parts until it is stopped explicitly.

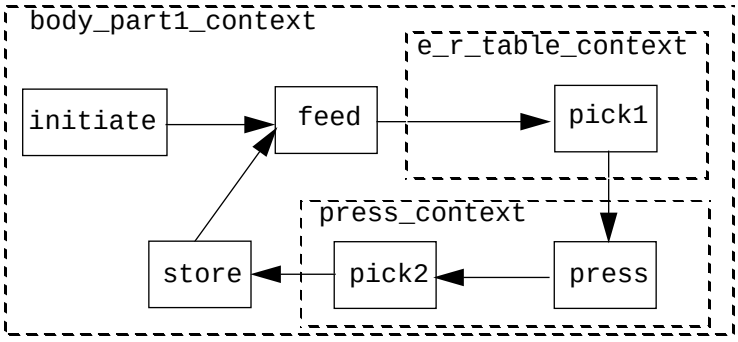


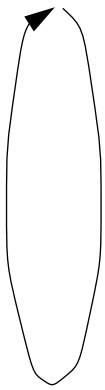
Fig. 5. An example of a task program for producing `part1`

Below follows part of an example schema in AMOSQL that illustrates the example above. The two main contexts are first defined followed by a context for the elevating rotary table. A rule that defines when the robot can grip a part on the table is activated into the context for the first arm of the robot (`:arm1`). A context for the press is then created along with a rule that specifies when it is safe to operate the press. The first rule is also activated into this context, but for the second arm of the robot (`:arm2`) instead. It specifies when the robot can grip an object in the press. Here follow extracts of the context related parts of the schema for this application:

```
create context body_part1_context;
/* Definitions of rules related to part1 */
...
create context body_part2_context;
/* Definitions of rules related to part2 */
...
create context e_r_table_context;
create rule grip_rule(robot_arm a) as
            when for each part prt
            where above(position(a), prt)
            do robot_grip(a, prt);
activate rule grip_rule(:arm1) into e_r_table_context;

create context press_context;
create rule press_rule(robot r, press p) as
            when for each robot_arm a
            where a = arm(r) and
                   outside(position(a), p) and
                   part_in_press_position(p)
            do close_press(p);
activate rule press_rule(:robot, :press) into
            press_context;
activate rule grip_rule(:arm2) into press_context;
```

During the execution the task program for producing `part1` the order of database operations initiated from the task level might be:



```
activate context body_part1_context;
...
            check(:body_part1_context);
            ...
            activate context e_r_table_context;
            ...
            check(:e_r_table_context);
            ...
            deactivate context e_r_table_context;
            ....
            activate context press_context;
            ...
            check(:press_context);
            ...
            deactivate context press_context;
```

5 Conclusions and Future Work

The paper presented rule contexts as a mechanism for dynamically grouping rules. Rules are activated into contexts and are deactivated from contexts. When a context is activated it enables all its rules for monitoring. In deactivated contexts all the rules are disabled from being monitored. Events are only monitored if they are referenced from some rule in an active context.

Contexts are used to represent coupling modes where all rules in the same context also share the same coupling mode. Predefined contexts are defined for the usual system coupling modes.

Contexts are first-class objects, which makes it possible to store them in any data structure and to define meta-rules that activate and deactivate them.

Future work includes investigating the need for several contexts belonging to the same coupling mode. This will cause a need for ordering the execution order of different contexts. Using priorities is one way of doing this, but since the conflict resolution between different rules inside the same context is also done with priorities this might lead to an unnecessary complicated model.

The issue of event consumption is also important. If checking of one context consumes events then rules in consecutively checked contexts might not trigger the way they were intended.

Defining meta-rules that manage other contexts is another subject for future research.

6 References

[1] Buchanan B. G. and Shortliffe E. H.: Rule-based Expert Systems, *The Mycin Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley, 1984.

[2] Davis R.: Meta-rules: Reasoning about Control, *AI*, vol. 15, 1980, pp. 179-222.

[3] Dayal U., Buchman A.P., and McCarthy D.R.: Rules are objects too: A Knowledge Model for an Active, Object-Oriented Database System, in *Proceedings of the 2nd International. Workshop on Object-Oriented Database Systems*, Lecture Notes in Computer Science 334, Springer-Verlag 1988.

[4] Dayal U. and McCarthy D.: The Architecture of an Active Database Management System, in *Proceedings of the ACM SIGMOD Conference*, 1989, pp. 215-224.

[5] Fahl G., Risch T., and Sköld M.: AMOS - An Architecture for Active Mediators, *International. Workshop on Next Generation Information Technologies and Systems (NGITS'93)* Haifa, Israel, June 1993, pp. 47-53.

[6] Fishman D. et. al.: Overview of the Iris DBMS, *Object-Oriented Concepts, Databases, and Applications*, ACM press, Addison-Wesley Publ. Comp., 1989.

[7] Georgakopoulos D., Hornick M., and Sheth A.: An Overview of Workflow Management: From Process Modelling to Workflow Automation Infrastructure, *Distributed and Parallel Databases*, 3, 2, April 1995, pp. 119-153.

[8] Hedberg S. and Steizner M.: Knowledge Engineering Environment (KEE) System: Summary of Release 3.1, Intellicorp Inc. July 1987.

[9] Loborg P., Holmbom P., Sköld M., and Törne A.: A Model for the Execution of Task-Level Specifications for Intelligent and Flexible Manufacturing Systems, *Integrated Computer-Aided Engineering* 1(3) pp. 185-194, John Wiley & Sons, Inc., 1994.

[10] Loborg P., Risch T., Sköld M., and Törne A., Active Object Oriented Databases in Control Applications, *19th Euromicro Conference of Microprocessing and Microprogramming*, vol. 38, 1-5, pp. 255-264, Barcelona, Spain 1993.

[11] Lyngbaek P.: OSQL: A Language for Object Databases, tech. rep. HPL-DTD-91-4, *Hewlett-Packard Company*, Jan. 1991.

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

[13] Risch T. and Sköld M.: Active Rules based on Object Oriented Queries, *IEEE Data Engineering bulletin*, Vol. 15, No. 1-4, Dec. 1992, pp. 27-30.

[14] Shipman D. W.: The Functional Data Model and the Data Language DAPLEX, *ACM Transactions on Database Systems*, 6(1), March 1981.

[15] Stonebraker M., Hearst M., and Potamianos S.: A Commentary on the POSTGRES Rules System, *SIGMOD RECORD*, vol. 18, no. 13, Sept. 1989.

[16] Tombros D., Geppert A., and Dittrich K. R.: SEAMAN: Implementing Process-Centered Software Development Environments on Top of an Active Database Management System, *Technical Report 95.03, Comp. Science Dept., University of Zürich*, Jan. 1995.

[17] Walters J.R. and Nielsen N.R., *Crafting Knowledge-based Systems - Expert Systems Made Easy/ Realistic*, John Wiley & Sons, 1988, pp. 253-284.

[18] Widom J.: The Starburst Rule System: Language Design, Implementation, and Applications, *IEEE Data Engineering*, vol. 15, no. 1 - 4, Dec. 1992.

15.6 Paper VI

M. Sköld and T. Risch: Using Partial Differencing for Efficient Monitoring of Deferred Complex Rule Conditions, presented at the 12th International Conference on Data Engineering (ICDE'96), New Orleans, Louisiana, February 1996.

This paper is incorporated into chapter 6.

15.7 Paper VII

L. Lin, T. Risch, M. Sköld, and D. Badal: Indexing Values of Time Sequences, presented at the Fifth International Conference on Information and Knowledge Management (CIKM'96), Rockville, Maryland, USA, November 12-16, 1996.

Indexing Values of Time Sequences

Ling Lin*, Tore Risch*, Martin Sköld*, Dushan Badal†

*Department of Computer Science
Linköping University, Sweden
{linli, torri, marsk}@ida.liu.se

†Department of Computer Science
University of Colorado at Colorado Springs, USA
badal@sunshine.uccs.edu

Abstract

A time sequence is a discrete sequence of values, e.g. temperature measurements, varying over time. Conventional indexes for time sequences are built on the time domain and cannot deal with *inverse queries* on a time sequence (i.e. computing the times when the values satisfy some conditions). To process an inverse query the entire time sequence has to be scanned. This paper presents a dynamic indexing technique on the value domain for large time sequences which can be implemented using regular ordered indexing techniques (e.g. B-trees). Our index (termed *IP-index*) dramatically improves the query processing time of inverse queries compared to linear scanning. For periodic time sequences that have a limited range and precision on their value domain (most time sequences have this property), the IP-index has an upper bound for insertion time and search time.

1 Introduction

In many real-time and temporal database applications the state of a data object *o*, varies over discrete time points, forming a *time sequence* (*TS*). A time sequence can be viewed as a state sequence S_i with $S_i=(t_i, v_i)$, where v_i is the value of the data object at time t_i.

There are three basic characteristics of such time sequences:

1. Time sequences are *ordered*, i.e. $\forall\ i, j:\ i > j \rightarrow t_i > t_j$.
2. Each value v_i is functionally dependent on the time t_i, but the inverse does not hold.
3. The value v_i can be 1-dimensional (e.g. for temperatures or voltages), 2-dimensional (e.g. for positions in a plane), or of higher dimensionality. In this paper we will concentrate on 1-dimensional data. The ideas presented can be extended to multi-dimensional data as well.

Two basic classes of queries on time sequences can be identified:

1. Forward queries, e.g.
   - What was the value at time point *t'?*
   - What was the value range in the time interval [*t', t''*]?
2. Inverse queries, e.g.
   - At what time point(s) t was the value equal to *v'*?
   - In what time interval(s) [*t', t''*] was the value larger (smaller) than *v'*?

Complex queries can be composed by combining these basic queries.

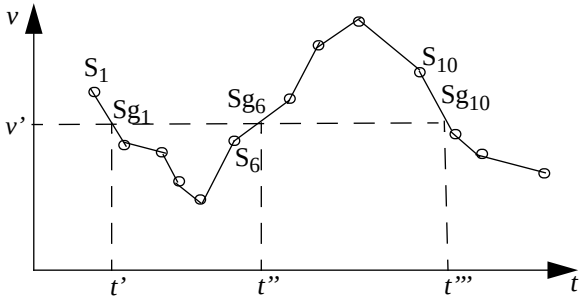


Fig. 1.1: Illustration of inverse queries

For example, if the data in Fig. 1.1 represents a patient's temperature reading over a time period, a forward query could be "What was the patient's temperature at 11:00 yesterday?", and an inverse query could be "At what time period did the patient have a temperature higher than 38°C?". Note that the result of the inverse query is a sequence of time intervals.

Forward queries can be supported by B+-trees[EWK93], AP-trees[GS93], I-trees[TMJ94], or by computational methods[F96]. Inverse queries, however, are difficult to support since there can be more than one time point (time interval) where the value is equal to (larger than, smaller than) *v'*. This paper provides an indexing method to efficiently answer inverse queries on *TS*s. The index supports efficient insertions of new states at the end of *TS*s.

The intuition behind our index is illustrated in Fig. 1.1. The *TS* is viewed continuously as a sequence of segments $Sg_i=[S_i, S_{i+1}]$. The time points when the value is equal to *v'* in Fig. 1.1 are <*t'*, *t''*, *t'''*>. These time points can be computed (by interpolation) if we get all the segments Sg_i that intersect the line *v=v'* (i.e., the segments Sg_1, Sg_6, Sg_{10} in Fig. 1.1). We propose an index method that retrieves all the intersecting segments for a value *v'*. This index performs especially well for periodic *TS*s with a limited range and precision on the value domain. We have measured its performance in a main-memory database.

2 Related Work

Much research has been done on time sequences. Most of it deals with similarity matches [AFS93][LYC96][SZ96], i.e., finding all similar time sequences (or subsequences) that match a given pattern within some error distance. Indexes [ALSS95][APWZ95] and query languages [APWZ95] have been developed to achieve this goal.

Several indexing methods have been proposed for temporal relations [EWK93][GS93][SOL94][TCGJSS93]. Most of them are intended to support operations like temporal join, temporal selection, etc., and they mostly assume interval time stamps rather than time points.

By contrast our goal is to develop an indexing technique to support inverse queries on *TS*s. This index can be seen as an index on the value domain rather than on the time domain. To the best of our knowledge no work has been done in this area.

Our idea is to transform the problem of inverse queries into k-dimensional spatial search problems, i.e. finding all intervals intersecting a given line. There have been several indexing methods proposed for k-dimensional spatial search, e.g. k-d trees [OMS87], R-trees [G84] and SR-Tree [KS91]. Some index trees have also been proposed in computational geometry to deal with interval problems, e.g., Interval Trees [E79], and Segment Trees [B72]. However, none of the above methods are suitable for inverse queries on *TS*s. The reasons are: 1) *TS*s consist of large sets of intervals $[S_i, S_{i+1}]$ which are dynamically growing, while most spatial data structures assume a fixed search space. 2) The intervals in *TS*s have a special property that the end point of Sg_i is the starting point of Sg_{i+1} (i.e. S_{i+1}). We will show that this property makes our index algorithm much more simple compared to R-trees. Our index method can be built upon a regular ordered one-dimensional index such as B-trees, while R-trees require a complicated algorithm for handling boundary conditions between regions.

Related work can be found in [EWK93] where temporal operations are viewed as interval intersection problems and where B^+-trees are used to index interval time stamps. Another related study [KS95] views temporal aggregation problems as an interval overlapping problem and then uses the Segment Tree [B72] to build an index for computing temporal aggregates.

[SS93] proposes a temporal data model for *TS*s. It defines four *types* of *TS*s according to what interpolation assumptions are applied, a) Step-wise constant (all values between $[S_i, S_{i+1}]$ are assumed to be equal to v_i), b) Continuous (a curve-fitting function is applied between $[S_i, S_j]$), c) Discrete (missing values cannot be interpolated) d) User-defined (a user-defined interpolation function is applied). Our index can be used to answer inverse queries covering all the above cases.

3 The IP-index

We start with the simplest inverse queries on *TS*s, i.e. “At what time point was the value equal to *v*?”, denoted as
$F^{-1}(v)$.

A naive way of answering inverse queries is to do curve fitting on the *TS* to generate the function $v=F(t)$, and then solve the equation $t=F^{-1}(v)$. This method is not practical when the *TS*s are long and dynamically growing.

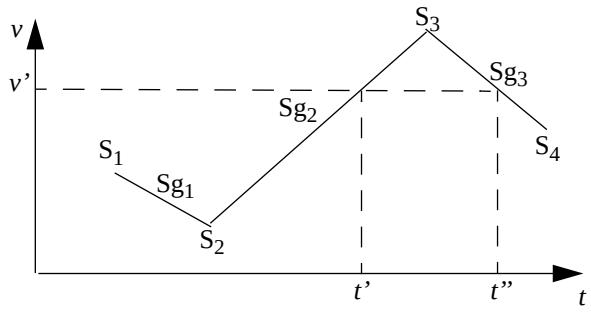


Fig. 3.1: An example *TS* and an inverse query

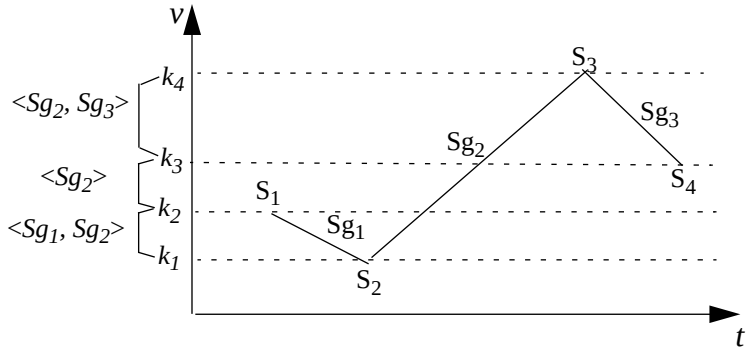


Fig. 3.2: Illustration of the IP-index

We propose a better solution. Each state S_i in the *TS* is viewed as points in the two-dimensional plane *t-v* as shown in Fig. 3.1. Then each consecutive states S_i, S_{i+1} constitute a line segment Sg_i. Then, if we can find all segments Sg_i that intersect the line $v=v'$, we can answer inverse queries. For example, in Fig. 3.1, the segments which intersect the line $v=v'$ are $<Sg_2, Sg_3>$. The answer of the inverse query $F^{-1}(v')$ will then be:

- If the "step-wise" constant or "discrete" assumption is applied, then $F^{-1}(v')=nil$, since there is no value defined between S_2, S_3 and S_3, S_4 respectively.

- If the "continuous" or "user-defined" assumption is applied, then $F^{-1}(v')=<t', t''>$, where t' and t'' are calculated by applying some interpolation function (e.g. linear interpolation, least square, etc.) on the states around the segments Sg_2 and Sg_3 respectively.

So, the problem of inverse queries is transformed into the problem of finding all the intersecting segments for the line $v=v'$. A naive way to solve the problem is to scan the entire *TS* to check if any two consecutive states S_i, S_{i+1} "contain" v', i.e. if $v_i < v' < v_{i+1}$, or $v_{i+1} < v' < v_i$. Such an algorithm, however, has the complexity of $\Theta(N)$, where N is the size of the *TS*. Below we propose an indexing technique to perform inverse queries more efficiently.

3.1 The IP-index Definition

If we project each line segment Sg_i on the v-axis, we get non-overlapping intervals $I_j=[k_j, k_{j+1})$, where each k_j is a distinct value of v_i (see $k_1...k_4$ in Fig. 3.2). We can see that all values that belong to one interval have the same sequence of intersecting segments (marked to the left in Fig. 3.2). Our index associates with each interval $[k_j, k_{j+1})$ all segments Sg_i that span[1] it. It is termed the *IP-index*. A simple illustration of the IP-index is shown in Fig. 3.2, where we associate each interval $[k_j, k_{j+1})$ with the sequence of spanning segments Sg_i.

Since the segments are consecutive, each segment Sg_i is uniquely identified by its starting state S_i. We use S_i to represent the segment Sg_i in the IP-index. We term the starting states of each segments that intersect the line $v=v'$ as the *anchor-states* of v'. Then, the sequence of intersecting segments can be represented as the sequence of anchor-states, which is termed the *anchor-state sequence*. The anchor-state sequence is a state sequence ordered by time.

Since each interval $[k_j, k_{j+1})$ is uniquely identified by its starting point k_j, we use k_j to represent the interval $[k_j, k_{j+1})$ in the IP-index.

Suppose that $k_1<k_2<...<k_j<...$ are the ordered distinct values of v_i in the *TS*. Then each index entry N_i in the IP-index has the form [*key, anchors*] where

- $N_i.key=k_j$.
- $N_i.anchors$ is the anchor-state sequence for all v' such that $v' \geq k_j$ and $v' < k_{j+1}$. It is also denoted as $anchors(k_j)$.

For example, the IP-index for the simple *TS* in Fig. 3.2 is:

$anchors(k_1)=<S_1, S_2>$

1. We say a segment Sg_i spans an interval I_i when the projection of Sg_i on the v-axis spans the interval I_i, i.e. if $Sg_i=((t_s, v_s), (t_e, v_e))$ and $I_i=(v_a, v_b)$, then $v_s \leq v_a$ and $v_e \geq v_b$.

anchors(k_2)=<S_2>
anchors(k_3)=<S_2, S_3>
anchors(k_4)=*nil*

We should point out that if the interpolation method introduces new extreme points (and thus introduces extra segments) to the original time sequence, the IP-index needs to be modified to include the extra segments as well.

3.2 The IP-index Insertion Algorithm

As we mentioned in the introduction, the IP-index supports efficient insertion of new states at the end of *TS*s. Each new state forms a new segment, and this section shows how to efficiently insert a new segment Sg_i into the IP-index.

Suppose that in Fig. 3.2 we have inserted states S_1, S_2 and S_3, and then we want to insert a new state S_4. This means that we already have three index entries in the IP-index with keys v_1 (=k_2), v_2 (=k_1), v_3 (=k_4) respectively, and we also have *anchors*(k_1)=<S_1, S_2>, *anchors*(k_2)=<S_2>, *anchors*(k_4)=*nil*. To insert the state S_4=(t_4, v_4) we need to do the following:

1. The new state S_4 creates a new index entry with the key v_4 (=k_3), which divides the existing interval [k_2, k_4) into two intervals, [k_2, k_3) and [k_3, k_4) (see Fig. 3.2).

   The segments that span the new interval [k_2, k_3) are the same as the segments that spanned the old interval [k_2, k_4) (which are already present in the IP-index), i.e., *anchors*(k_2)=<S_2> stays unchanged.

   The segments that span the new interval [k_3, k_4) are the segments that spanned the old interval [k_2, k_4) plus the new segment Sg_3, i.e.,
   anchors(k_3)= *anchors*(k_2)+[1]S_3 = <S_2>+S_3 = <S_2, S_3>.

2. For all the entries in the IP-index with keys inside the interval [k_3, k_4) (in Fig. 3.2 there happens to be no such key), append S_3 to the end of their associated anchor-state sequences. This is because Sg_3 spans all the sub-intervals inside the interval [k_3, k_4).

The result of the insertion conforms with Fig. 3.2.

So, the insertion of a new state S_i=(t_i, v_i) (i=4 in the above example) into the IP-index has the following steps:

1. If v_i is an existing key in the IP-index, then go to step 4.
2. Search the index entries N_i in the IP-index to find the index entry N_L where $N_L.key = max\{(N_i.key) \mid ((N_i.key) \leq v_i)\}$.

1. We use '+' to denote adding a new element to the end of a sequence.

This step finds the existing interval (in the example of Fig. 3.2 we have $N_L.key=k_2$ thus the interval is $[k_2, k_4)$) which the new key (v_4 in the example) lies within.

3. Insert a new index entry with $key=v_i$ and *anchors* $=N_L.anchors$.

4. For all index entries N_j where $N_j.key$ lies inside the interval $[\min(v_{i-1}, v_i), \max(v_{i-1}, v_i))$, append the starting state (S_{i-1}) of the new segment to $N_j.anchors$.

3.3 Implementation

Note that the above insertion algorithm is about how to associate the intersecting segments with each inserted value v_i. It does not assume any specific implementation. Actually the IP-index can be implemented by any ordered indexing technique, e.g., B-Trees, AVL-Trees[AL62] or 2-3 Trees[C79], and the anchor-state sequence can be implemented as a sequential data structure (list or array) which supports fast appending.

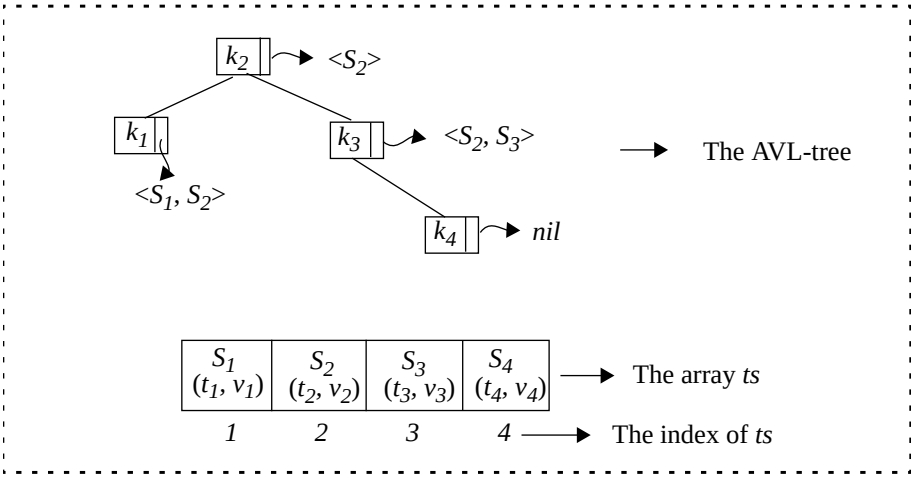


Fig. 3.3: The AVL-tree implementation of the IP-index in Fig. 3.2

To verify our ideas we implemented the IP-index in a main-memory database [FRS93]. The time sequence was stored in an array *ts*, where *ts*[i]=(t_i, v_i). We used an AVL-tree as indexing data structure since it has small re-balancing time. (Notice that the keys v_i do not arrive in order, which means that the tree needs to be re-balanced during insertion.) Each *index entry* in the above algorithm corresponds to a *node* in the AVL-tree. The anchor-state sequence was implemented as a dynamic array of integers, where each integer is an index of the array *ts*.

Fig. 3.3 illustrates the AVL-tree implementation of the IP-index in Fig. 3.2.

Before we give the insertion algorithm for the AVL-tree implementation of the IP-index, we explain the notation and functions that will be used in the algorithm:

- *tree* -- the AVL-tree implementing the IP-index.
- *ts* -- the array storing the time sequence
- $S_i=(t_i, v_i)$ is the arriving state (to be inserted into the IP-index).
- *insert_ts(ts, i, S_i)* -- inserts the state S_i into array *ts* where *ts*[*i*]=(t_i, v_i).
- *insert_node*(*tree*, v_i, *anc)* -- inserts into the AVL-tree a new node with key=v_i, *anchors=anc*.
- *get_lower*(*tree*, v_i): searches the AVL-tree to find the node N_L where

  $N_L.key = max\{N_j.key \mid N_j.key \le v_i,\ 1 \le j \le size(tree)\}$

  This function is used to find the existing interval which needs to be split into two parts; e.g. in Fig. 3.2, *get_lower*(*tree*, v_4).*key*=k_2. The function returns *nil* if no node is found.
- *exist_key(tree, v_i)*: returns *true* if there already exists a node in the IP-index whose key is equal to v_i.

The code for the IP-index implementation using an AVL-tree is as follows:

```
Insert_ip(tree,ts,t_i,v_i):
    S_i=<t_i,v_i>
    insert_ts(ts,i,S_i)
      /* insert the state into the array which stores the time sequence */
    if not exist_key(tree,v_i)
      N_L=get_lower(tree,v_i)                    (part 1)
      if  N_L=nil
        insert_node(tree,v_i,nil)
      else
        insert_node(tree,v_i,N_L.anchors)
          /* insert a new index entry, and copy the anchor state sequence
             from the "lower" index entry */
      endif
    endif
    if i>1
      /* if not the first state in the time sequence */
      for each node N_j                          (part 2)
              (1≤j≤size(tree))
        where N_j.key lies inside the interval [min(v_i-
          1,v_i),
          max(v_i-1,v_i))
        N_j.anchors=N_j.anchors+(i-1)
```

```
            /* add the new anchor state to the anchor state sequences of all the
               intervals spanned by the new segment */
        end for each
    endif
```

Fig. 3.4: The IP-index insertion algorithm

3.4 Performance vs. Precision of Values

This section discusses the relationship between the performance of the IP-index and the precision of the values in the time sequence.

The algorithm in Fig. 3.4 contains two parts. Algorithm analysis shows that (Part 1) takes $\Theta(LogM)$ time where M is the total number of index entries in the IP-index, since they are actually AVL-tree search operations. Furthermore, (Part 2) takes $m*append\_time$ where m is the number of intervals which are spanned by the new segment (*append_time* is the time taken to add the new state to the end of an anchor-state sequence, which is assumed to be constant since we use a sequential data structure which supports fast appending).

So, if we limit the parameters M and m, we can reduce the insertion time of the IP-index. This can be achieved by limiting the precision of the measured values. The reason is: for a time sequence with range=R and precision=P in the value domain, the number of index_entries will be less than R/P. So, the lower the precision (the larger the value of P) is, the smaller the value of M and m will be. Thus, we can reduce the insertion time by limiting the precision of the values, which will be shown in the performance measurement section.

The above observation is practical since 1) all measured time sequences have a limited range on value domain, 2) the original precision of the measured data is always limited due to errors and uncertainty in measurements. For example, when measuring the temperature of a patient the value range is the temperatures that the human body can possibly be alive at and at a precision that can represent changes that affect the well-being of the patients. Therefore, even if the thermometer used for measuring the temperature of a patient has the precision of 0.001°C, we can still limit the precision to 0.1°C, which will both improve the performance of the IP-index and still be reasonable for the application.

From the above discussions we can see that the IP-index is not suitable for some unusual time sequences, e.g. periodic time sequences with unlimited precision, or signals which oscillate with an increasing amplitude over time (these two kinds of time sequences have unlimited range or precision, which makes the M parameter large). It is also not suitable for those time sequences with “big jumps” in the values all the time (this kind of time sequence makes the parameter m large). Fortunately, most time sequences from real applications do not have these properties.

3.5 The IP-index Search Algorithm

To search the IP-index is to find the index entry which contains the anchor-state sequence of the value v', i.e., to search the index entries N_i in the IP-index to find the

index entry N_L where

$$N_L.key = max\{(N_i.key) \mid ((N_i.key) \leq v')\} \quad (1)$$

Then $N_L.anchors$ contains the anchor-state sequence for the value *v'*.

In the example *TS* in Fig. 3.1, the index entry N_L for the value *v'* is [k_3, *anchors*] where *anchors*=<S_2, S_3>. The reason is that k_3 (=v_4) is the first key which is "below" or equal to *v'*.

The search algorithm is dependent on the implementation of the IP-index. In the AVL-tree implementation the search algorithm is to search for the node in the AVL-tree whose key satisfies the above condition. It is well known that the complexity is $\Theta(LogM)$ where *M* is the total number of nodes in the tree.

As we discussed in the last section, the value of the parameter *M* is determined by the precision of the values. The lower the precision is, the smaller the value of *M* will be. So, limiting the precision of values will reduce the search time of the IP-index as well.

4 Queries

There are several kinds of queries that can be answered efficiently using the IP-index. Using an example of a patient's temperature reading, we can answer queries like:

- Query 1: When did the patient have the temperature 37°C?

  This query is expressed as $F^{-1}(37)$ and it represents the simplest form of inverse queries. It only requires searching the IP-index to find the anchor-state sequence for the value 37 (plus some post-processing if interpolation is needed).

- Query 2: During what time period did the patient have the temperature higher than 37°C?

  This query can be expressed as $F^{-1}(v>37)$. We refer this kind of query as *inequality inverse queries*. To answer this query we first calculate all time points equal to
  $F^{-1}(37)$: These time points form a sequence of time intervals. Then for each time interval we check if the values inside the interval are greater than *37* or not. If so, then this time interval is returned.

- Query 3: When did the patient have a temperature *around* 37°C?

  This query can be expressed as $F^{-1}((37-e, 37+e))$, while *e* is a value which is application dependent. This kind of query is useful since many applications need to know "When was the value *approximately* equal to *v'*?" rather than "When was the value *exactly* equal to *v'*?".

Query 3 can be easily computed given that we can compute Query 2. This is because
$F^{-1}((v', v'')) = F^{-1}(v > v') \cap F^{-1}(v < v'')$, where '$\cap$' means 'interval intersection' and $v'=37-e$ and $v''=37+e$.

- Query 4: When did the patient have two consecutive fevers during 24 hours?

  This is used for analysing the symptoms of disease[SZ96]. It is an example of shape queries on *TS*s. It can be computed as follows: 1) compute $F^{-1}(v>37)$ (which are the periods of "fever"), 2) check if there exist two time intervals in the "fever" periods that have the distance *d* of *24 hours*. (The distance between two time intervals can be defined either as the distance between the starting points of both intervals or as the distance between the mid-points of both intervals.)

5 Performance Measurements

We tested the performance of the IP-index using the AVL-tree implementation in a main-memory database[FRS93][1].

We measured the insertion time and search time of the IP-index for different kinds of *TS*s. The size (number of states) of each *TS* was 10000.

1. A simulated periodic sequence, sin(t/100) (t=1...10000) with very high precision, plotted in Fig. 5.1.
2. An application time sequence[JZBSL96] (which is the measurement of pressure in a fluidized bed) plotted in Fig. 5.2.
3. A simulated time sequence with a largely monotonic trend (but not strictly monotonic) plotted in Fig. 5.3.

1. All measurements were made on an HP9000/710 with 64 Mbyte of main memory and running HP/UX.

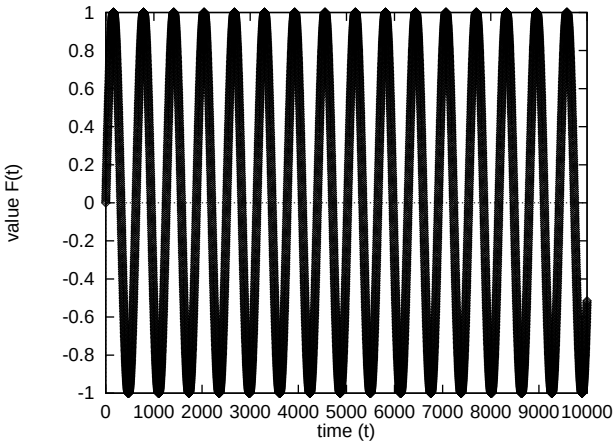


Fig. 5.1: Sinus Data

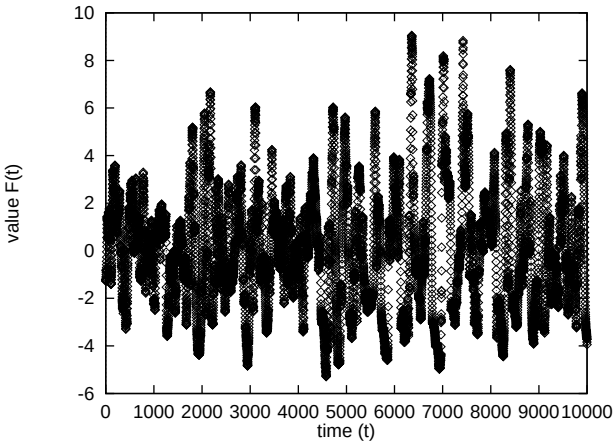


Fig. 5.2: Pressure Data

5.1 Insertion for Periodic Time Sequences

In Fig. 5.4 and Fig. 5.5 we show the measured insertion times of the IP-index for the time sequences shown in Fig. 5.1 and Fig. 5.2 respectively. The insertion times are measured as the sequences grow.

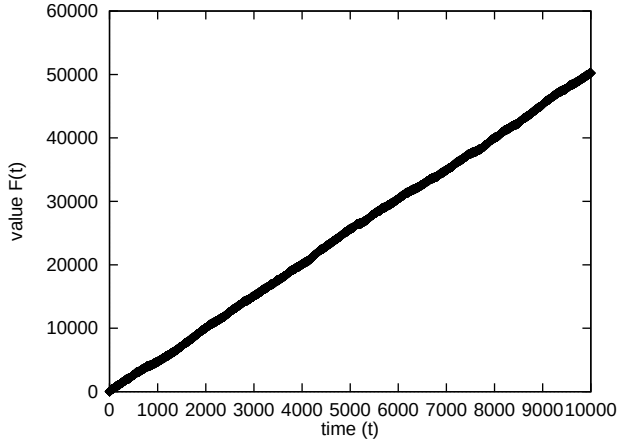


Fig. 5.3: Monotonic Trend Data

- The curves labelled “Original Value Insert” show the insertion times of the IP-index. For the pressure data the range=[-6, 10] and the precision=0.001. For the sinus data the range=[-1, 1] and the precision= 0.000001. We can see that the insertion time increases linearly with the size of the sequence. This is because the precision is very high which makes the parameters *M* and *m* (see section 3.4) large.

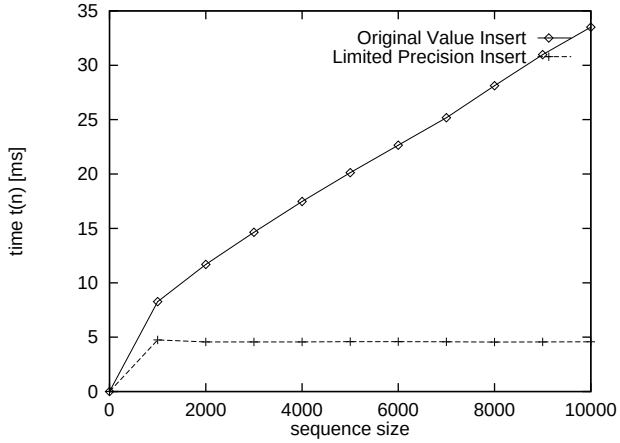


Fig. 5.4: Sinus Data Insertion

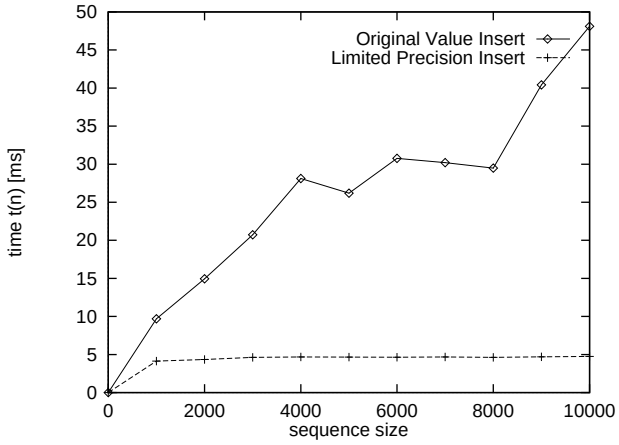


Fig. 5.5: Pressure Data Insertion

- For the curves labelled “Limited Precision Insert” the precision of the values is limited to 0.1 for the pressure data and 0.001 for sinus data respectively. We notice that the insertion time become constant after the total number of index entries has been inserted into the IP-index. This is because a) the limited precision makes the number of nodes of the AVL-tree does not grow any more; only the anchor-state sequence associated with each node grows with the time sequence and b) the limited precision makes the *m* parameter (number of intervals spanned by the new segment as discussed at the end of section 3.4) have an upper limit, which causes the insertion time to have an upper limit (See Fig. 3.4 (Part 2)).

Our measurements verify that for a periodic time sequence with a limited range and precision on the value domain, there will be an upper bound on the IP-index insertion time.

5.2 Search for Periodic Time Sequences

In Fig. 5.6 the IP-index search time is compared with linear scanning of the time sequence to find the anchor-state sequence for some randomly generated value *v’*. The test was done on the simulated periodic sequence with very high precision as plotted in Fig. 5.1. The comparison was measured as the sequence grows. The results show that the IP-index dramatically improves the performance of inverse queries. Note that the results are displayed in logarithmic scale since the difference between the IP-index search time and the linear scanning time is too great to display with linear scaled axes. (Note that the curve labelled “IP-index Search” in Fig. 5.6 is the same as the one labelled “Original Value Search” in Fig. 5.8; they do not look the same because they are displayed on different scaled axes.)

Fig. 5.8 and Fig. 5.7 show the performance of the IP-index search for two periodic *TS*s. After every 1000 insertions, we measured the average time to search for the anchor-state sequence for some randomly generated value *v’*. The results show that

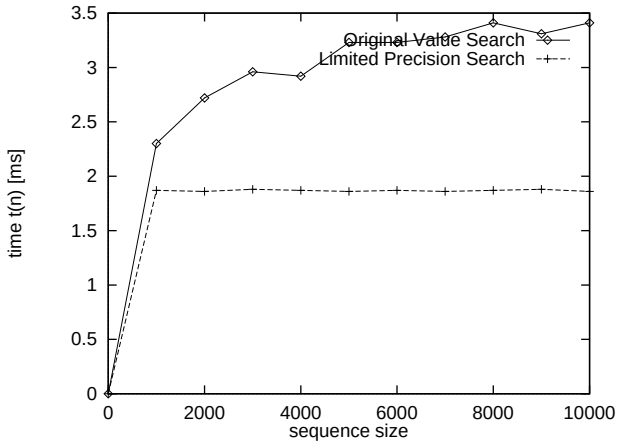


Fig. 5.6: Compare the IP-index with Linear Scanning

the search time is logarithmic due to the AVL-tree implementation (see the curves labelled “Original Value Search”). However, when the assumption of “limited range and precision” is satisfied, the IP-index search time has an upper bound regardless of the time sequence size (see the curve labelled “Limited Precision Search”). The reason is the same as in the insert case, i.e., the number of nodes (M) of the AVL-tree does not increase after all index entries are inserted, (only the anchor-state sequences associated with each node grow) so the search time stays constant to $\Theta(LogM)$.

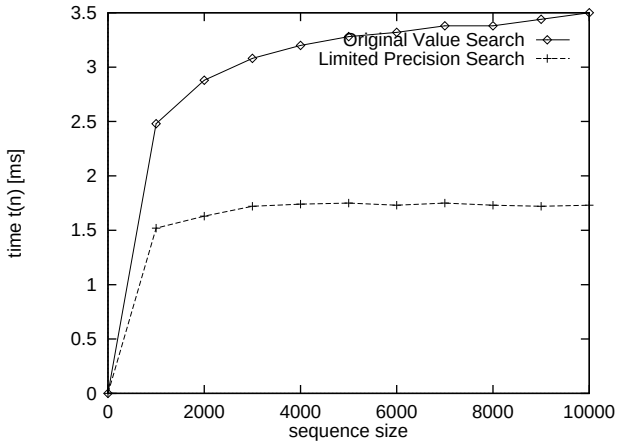


Fig. 5.7: Pressure Data Search

5.3 Time Sequences with Monotonic Trends

In Fig. 5.9 we measured the performance of the IP-index for a simulated time sequence with a largely monotonic trend. We see that both the insertion time and the search time are approximately logarithmic due to the AVL-tree implementation. Since in this case we do not have a limited range on the value domain, the "upper limit" on insertion and search time cannot be achieved.

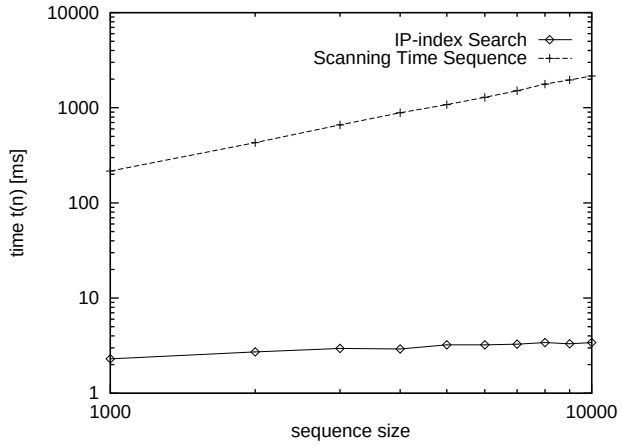


Fig. 5.8: Sinus Data Search

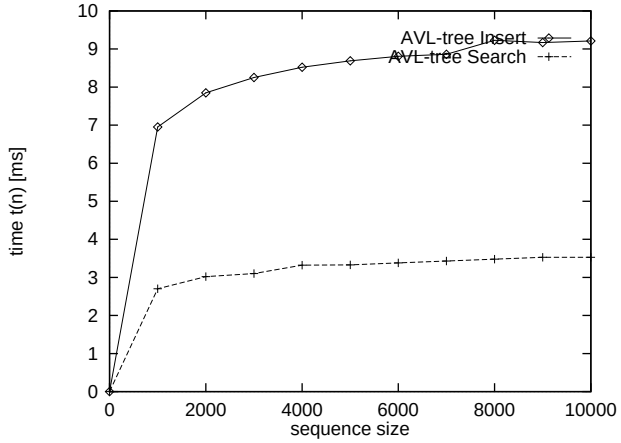


Fig. 5.9: Monotonic Trend Data

We also notice that a strictly monotonic time sequence does not need any IP-index. The reason is that the value domain is then monotonic just as the time domain is,

which means that conventional indexes on the time domain can be applied to the value domain.

6 Conclusions and Future Work

This paper presented the IP-index which is an index on the value domain for time sequences. We showed how to use the IP-index to support *inverse queries*, such as finding all the time points when the temperature was equal to a given value *v* (computing $F^{-1}(v)$), or to find the time intervals where the values are greater (smaller) than a given value *v'* (computing $F^{-1}(v>v')$ or $F^{-1}(v<v')$).

The IP-index can be implemented using any ordered index structures, such as B-trees. The performance measurements showed that the IP-index radically improves the processing time of inverse queries on time sequences, compared to linearly scanning the sequence (the only alternative without the IP-index). For a periodic time sequence with a limited range and precision on the value domain, the IP-index insertion and search time have an upper bound regardless of the size of the sequence. Furthermore, by limiting the precision of the values the IP-index insertion and search times can be dramatically improved.

In future work we will investigate how to use the IP-index in query optimization. For example, we can define the cost models for the IP-index and store the cardinality (the lengths of the anchor-state sequences) in each index entry in order to estimate the cost of $F^{-1}(v')$ when the *TS* is very long. We can also set a "moving window" on the anchor-state sequence to discard or archive the old values of the *TS* when they are not required any more.

Another improvement is to extend the IP-index for indexing collections of *TS*s [SS93] based on the composite key *o*+*v* (*o* is the identifier of each *TS*).

Another future study will be to generalize the IP-index to n-dimensional *TS*s, e.g. to query the past positions of an aircraft given that the *TS* is a spatial-temporal trajectory of the aircraft.

We also need to explore the IP-index for very large time sequences stored on disk.

7 Acknowledgements

The authors would like to acknowledge Olof Johansson for the valuable discussions which led to the IP-index concept.

References

[AL62] G. M. Adelson-Velskii and E. M. Landis: *Doklady Akademia Nauk SSSR*, 146, (1962), pp. 263-266; English translation in *Soviet Math*, 3, pp. 1259-1263.

[AFS93] R. Agrawal, C. Faloutsos, and A. Swami: Efficient Similarity Search in Sequence Databases. In *Proceedings of the Fourth International Conference on Foundations of Data Organization and Algorithms*, pp. 69-84, Chicago, Oct. 1993.

[ALSS95] R. Agrawal, K. Lin, H. S. Sawhney, and K. Shim: Fast Similarity Search in the Presence of Noise, Scaling, and Translation in Time-Series Databases. In *Proceedings of the VLDB, Conference*, pp. 490-501, 1995.

[APWZ95] R. Agrawal, G. Psaila, D. L. Wimmers, and M. Zaït: Querying Shapes of Histories. In *Proceedings of the 21st VLDB Conference*, pp. 502-514, 1995.

[B72] J. L. Bently: *Algorithms for Klee's Rectangle Problems*, Computer Science Department, Carnegie-Mellon University, Pittsburgh, 1972.

[BWBJ95] C. Bettini, X. S. Wang, E. Bertino, and S. Jajodia: Semantic Assumptions and Query Evaluation in Temporal Databases. In *Proceedings of the SIGMOD Conference*, pp. 257-268, May 1995.

[C79] D. Comer: The Ubiquitous B-Tree. In *ACM Computing Surveys*, 11, 2, pp. 121-137, June 1979.

[E79] H. Edelsbrunner: *Dynamic Rectangle Intersection Searching*, Institute for Information Processing, Rept. 47, Technical University of Graz, Graz, Austria.

[EWK93] R. Elmasri, G. T. J. Wuu, and V. Kouramaijian: The Time Index and the Monotonic B^+-tree. In [TCGJSS93], pp. 433-455.

[F96] E. T. Falkenroth: Computational Indexes for Time Series. In *Proc. of 8th Intl. Conference on Scientific and Statistical Database Management*, pp. 18-23, June 1996, Stockholm, Sweden.

[FRS93] G. Fahl, T. Risch, and M. Sköld: An Architecture for Active Mediators. In *Proceedings of the International Workshop on Next Generation Information Technologies and Systems*, pp. 47-53, Haifa, Israel, 1993.

[GS93] H. Gunadhi and A. Segev: Efficient Indexing Methods for Temporal Relations. In *Transactions on Knowledge and Data Engineering*, Vol. 5, No. 3, pp. 496-509, June 1993.

[G84] A. Guttman: R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the ACM SIGMOD Conference*, Boston, MA, June 1984.

[JZBSL96] F. Johnsson, R. C. Zijerveld, C. M. van den Bleek, J. C. Schouten, and B. Leckner: *Characterization of Fluidization Regimes in Circulating Fluidized Beds - time series analysis of pressure fluctuations*, Technical Reports, Chalmers Institute of Technology, Sweden, 1996 (submitted for publication).

[KS95] N. Kline and R. Snodgrass: Computing Temporal Aggregates. In *Proceedings of the Data Engineering Conf.*, pp. 222-231, 1995.

[KS91] C. P. Kolovson and M. Stonebraker: Segment Indexes: Dynamic Indexing Techniques for Multi-Dimensional Interval Data. In *Proc. ACM SIGMOD Conference*, pp. 138-148, 1991.

[LYC96] C. S. Li, P. S. Yu, and V. Castelli: HierachyScan: A Hierachical Similarity Search Algorithm for Databases of Long Sequences. In *Proceedings of the Data Engineering Conference*, pp. 546-553, Feb. 1996.

[OMS87] K. Ooi, B. McDonell, and R. Sacks-Davis: Spatial kd-tree: Indexing mechanism for spatial database. In *IEEE COMPSAC 87*, 1987.

[SS93] A. Segev and A. Shoshani: A Temporal Data Model Based on Time Sequences. In [TCGJSS93], pp. 248-269.

[SZ96] H. Shatkay and S. B. Zdonik: Approximate Queries and Representations for Large Data Sequences. In *Proceedings of the Data Engineering Conference*, pp.536-545, Feb. 1996.

[SOL94] H. Shen, B. C. Ooi, and H. Lu: The TP-Index: A Dynamic and Efficient Indexing Mechanism for Temporal Databases. In *Proceedings of the Data Enginee-*

ring Conference, pp. 274-281, 1994.

[TCGJSS93] A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass (editors): *Temporal Databases, Theory Design and Implementation*, The Benjamin/ Cummings Publishing Company, ISBN 0-8053-2413-5, 1993.

[TMJ94] K. Torp, L. Mark, and C. S. Jensen: *Efficient Differential Timeslice Computation*, Technical Report, College of Computing, Georgia Institute of Technology, Georgia, USA, Sept. 1994.

Index

LaTeX style references for paper VII, not part of thesis!!

AL62
AFS93
ALSS95
APWZ95
B72
BWBJ95
C79
E79
EWK93
F96
FRS93
GS93
G84
JZBSL96
KS95
KS91
LYC96
OMS87
SS93
SZ96
SOL94
TCGJSS93
TMJ94