Active Database Behaviour

The REFLEX Approach

Waseem Hadder Naqvi

A thesis submitted in partial fulfilment of the requirements of the University of Greenwich for the degree of Doctor of Philosophy

June 1995

University of Greenwich

London

Acknowledgements

I would like to express my thanks to a number of people who, during the course of this research, have strived to make the process more humane.

Firstly, Mohamed T. Ibrahim, my Supervisor and Director of Studies, for his advice and guidance during the past years but above all for his friendship. Professor Brian Knight who, even though involved in the latter stages of this research, proved an invaluable source of advice and guidance. Yasser Ades of course must get a mention, as he has never reconciled the fact that I was not researching in semantic analysis. Dr. Ala Al-Zobaidie for his encouragement and advice throughout the years. A special thank you to Malcolm Hudson for his endless support, friendship and belief.

I would sincerely like to thank David Marsh and Geoff Cooper, my present employers at the Computer Services Centre, University of Greenwich, for allowing me the time to complete this thesis.

I would like to thank my colleagues Colin Hughes, Diane Gan, Steve Panyioutou, Ahmed Alyamani, Robert Leslie, Dr. Sati McKenzie, Dr. Don Cowell and Dr. Garret Kearney, for their friendship and for the useful discussions during the course of this research.

I would like to thank my father Sayed Akthar Ali Naqvi, Shaheen, and my friends and colleagues for their support and encouragement over the years. Especially Farah for her patience in draft reading the thesis and Faraaz for not allowing me to work on weekends thus preserving some sanity and for the use of his computer.

I would also like to acknowledge the numerous people that I have met in the course of this research for both making the process interesting and for providing motivation.

Abstract

Modern day and new generation applications have more demanding requirements than traditional database management systems (DBMS) are able to support. Two of these requirements, *timely* responses to the change of database state and *application domain knowledge* stored within the database, are embodied within active database technology.

Currently, there are a number of research prototype active database systems throughout the world. In order for an organisation to use any such prototype system, it may have to forsake existing products and resources and embark on substantial reinvestment in the new database products and associated resources and retraining costs. This approach would clearly be unfavourable as it is expensive both in terms of time and money.

A more suitable approach would be to allow active behaviour to be added onto their existing systems. This scenario is addressed within this research. It investigates how best active behaviour can be augmented to existing DBMSs, so as to preserve the investments in an organisations resources, by examining the following issues, (i.) what form the knowledge model should take, (ii.) should rules and events be modelled as first class objects, (iii.) how will the triggering events be specified, (iv.) how will the database state be tested, (v.) how will resultant actions be executed, and (vi.) how the user will interact with the system.

Various design decisions were taken, which were investigated by implementation of a series of working prototypes, on the ONTOS DBMS platform. The resultant REFLEX model was successfully ported and adapted onto a second POET platform. The porting process uncovered some interesting issues regarding preconceived ideas about the portability of open systems.

Contents

Appendices

List of Figures

List of Tables

Chapter 1

Introduction

The contents of this thesis report the results of an investigation into how existing commercial organisational database management systems can be extended with the ability to utilise an active knowledge management system, by considering the following issues, what form the knowledge model should take, should rules and events be modeled as first class objects, how will the triggering events be specified, how will the database state be tested, how will resultant actions be executed, and how the user will interact with the system. The research has concentrated on augmenting an object oriented (OO) database system with active behaviour. The main objectives are to identify, represent and extend an existing database with active behaviour, allowing the encoding of domain knowledge within the host database management system in an efficient manner.

1.1. Motivations and Contribution of the Research

A database stores information about some *part of the real world*, sometimes referred to as the *miniworld* or the *universe of discourse* (UoD). Many applications such as process control, computerised stock/securities trading and network management require timely responses to critical situations, as observed by Dittrich et al. [Dittrich 86].

These applications are not well served by passive database management systems (DBMS), where actions are performed on the database by user or program requests, since these databases are simple repositories of data without any knowledge of what the data is to be used for. For the purpose of this research we shall refer to these databases as *traditional databases*. The data once entered into a database of this kind, may cause the system to be in a *semantically inconsistent* state i.e. the internal database state may not truly represent the external real world. For example in a Students Record System, on the death of a student it is meaningless to have the student still enrolled at the university. This situation may be rectified by an application which has knowledge as to how to reset the internal state of the database to the external real world state, by polling the database at the prescribed period. The interval between polling periods may be large, after which the relevance of the data may be in doubt. The interval between the polling of the database may be reduced causing the database to be polled more frequently. This approach causes increased overhead as the database is serving polling requests rather than serving its intent. A frequently used alternative strategy is to augment the code, within the application, which updates the database with additional logic to test any repercussions of the data entered. This has severe consequences for system maintenance since the code required to maintain the semantic integrity would have to be duplicated among the many programs which access a particular item of data. Even if the system was implemented using a modular approach, the code is still replicated using cut and paste techniques, each of which need to be changed to reflect any new knowledge. The code redundancy has to be maintained which leads to large maintenance costs.

A solution to the problem of code redundancy is to model this domain knowledge within the database. However, several authors have drawn a distinction between knowledge and data [Freundlich 90, Ibrahim 95, Luger 89, Ringland 87]; knowledge being represented in various forms. Today, many commercial database systems provide this support for knowledge in the form of integrity constraints, which are a mechanism to help preserve the semantic integrity of the database system. They allow some

knowledge to be attached to the system. This support by means of integrity constraints is realised by a simple collection of triggers (current DBMSs which support triggers include ORACLE, INGRES, and Sybase). However, there is much more domain knowledge that an application designer would like to support, for which trigger mechanisms are inadequate. For example, as pointer out by Stonebraker et al. [Stonebraker 89], one might want to insist that a specific employee, Nigel, has the same salary as another employee, John. This rule would be difficult to enforce in application logic because it would require the application to see all updates to the salary field, in order to fire application logic to enforce the rule at the correct time. A better solution would be to enforce the rule inside the DBMS.

In active database systems, the data, knowledge and parts of the processing logic (relating to events and conditions that require action) are under the control of an *active* database management system (ADBMS).

An apt definition for active databases is that provided by Medeiros and Pfeffer [Medeiros 90], they state:

> "Active databases differ from conventional databases as they are enhanced with *active* behaviour, i.e. behaviour exhibited *automatically* by the system in response to events generated internally or externally without user intervention".

Active databases respond automatically to any given events, but how is this knowledge encoded within the system? According to McCarthy and Dayal [McCarthy 89], the user may provide knowledge in the form of Event-Condition-Action (ECA) production rules. The ECA rules are akin to the production rules found in expert systems with the addition of an event clause. The rules are made up of three parts, (i.) an event clause (ii.) a condition clause and (iii.) an action clause. Once these rules have been defined, the system, on the change of database state, or other external events, evaluates the

condition(s) of any triggered rule. If the condition has been satisfied, it then without user intervention, executes the action clause of the rule. It does not need to wait for either user or program invocation as with a passive system.

Since active databases can respond to a given situation almost as it occurs, they would be of great use in situations where any changes to the data are of paramount importance and that the changed database state is acted upon immediately as severe penalties may be incurred as time elapses, i.e. real-time. Examples of real-time databases could be air-traffic control systems, computer aided manufacturing systems and many process control applications.

Some other more typical day-to-day examples could be administrative systems such as university Student Record Systems (SRS) or company Payroll applications. These applications are prone to change for example, in an SRS the business rules are continually changing from semester to semester, as new courses (or at least course offerings) are initiated. The entry requirements for these courses may change from one session to the next and more importantly, so do the assessment criteria.

Active databases introduce further problems of activity or knowledge design, akin to the problems of expert systems design. In many cases the semantics of the problem domain are simply not well known. For example, in the case of an SRS, in terms of the assessment criteria, what are the conditions that must apply so that a student can *'pass'*? Who knows? Are they the same as last year? Are they formally recorded somewhere? Are they built into some system whereby the students grades are entered, and depending on the total marks, the student is awarded a degree or not? Or are they simply in the head of some administrator? To answer these questions one must apply some knowledge elicitation (KE) techniques.

1.1.1. Research Aims

Not only are active databases and the requirement to encode more domain knowledge in a centralized database an extremely interesting research area, but they are becoming increasingly important practically.

Previously, the related research work in this area was undertaken by either the creation of new DBMSs or by the *substantial re-engineering of existing DBMSs*. For a commercial organisation that requires that its knowledge is integrated within its DBMS systems, the above mentioned approaches are not suitable in terms of capital, time and confidence in a new system, especially when the system is of strategic importance, as is a DBMS. A favourable approach would be to allow their existing system to be augmented with active behaviour. This approach would allow a known DBMS (to the organisation) to become active, and thus result in cost savings in both resources and training. Since the staff would be familiar with the host database, these skills would be preserved. Corporations make substantial investments in applications software, which do not become evident until a few years later. This situation is made more acute when corporations intent on preserving their production systems (which very quickly, even in these days of supposedly open systems, become dinosaurs or legacy systems [Brodie 93]), discover that they are tied to a particular platform. Hence they cannot migrate to a different platform even if they wish to.

This was the prime motivation for this research, raising the question as to whether active functionality can be *bolted-on* to existing commercial databases and if so, *how best it could be accomplished*. This facility, which subscribes to the notions of open systems and the inherent portability that they offer, differs from the work of existing active database research, where the researchers have concentrated on building systems from scratch or at least where they have had access to the source code of their host system.

This research attempts to ascertain how best an active database should be structured and managed so that it coexists and adapts to its host DBMS and allows the domain knowledge to be represented explicitly in an object DBMS.

In order to achieve the main goal of the research, a number of pertinent issues must be considered, the answers to which can be found in chapters five and six:

i. what form should the knowledge model take?

ii. should events be modeled as first-class objects, or attributes of rules? What about composite events, should they be modeled as first-class objects?

iii. how should triggering event(s) be specified and evaluated?

iv. how are conditions on the state of the database to be specified and evaluated efficiently?

v. how will a user interact with the active database system, i.e. issues of human-computer interaction (HCI) require consideration ?

These questions and further questions are addressed throughout this thesis, and help to define a best/optimal active model. The results or findings of the research are the REFLEX active database model and are embodied in the various REFLEX active database prototypes.

1.2. Research Methodology

The problem domain was critically investigated with respect to related work and is

described later in chapter three, the theoretical solutions to the problems or issues in question were formed and are reported in chapter four. The primary question involved the manner of augmenting an existing commercial database with active functionality. In order to prove the theory, it was necessary to construct a prototype, embodying the proposed solutions. During the building and execution of the prototype, further questions and issues were raised. These were then tackled theoretically and the best solutions were implemented in further prototypes, repeating the cycle.

This method was adopted since a pure theoretical analysis can often miss out some features because they may be obvious or minimal, but, could be a crucial part of any model. The building of a prototype helped to realise the specific goal and provided useful feedback into the research investigation.

The use of standard examples during the design and testing phase of the prototype implementation and during the writing of this thesis allowed the train of thought a degree of coherency. The examples were i) an administrative system, a students records system (SRS) and ii) real-time, an air traffic control system (ATCS). Both of the example scenarios are fully described in the appendices.

The notation employed in this thesis for the representation of objects is that of Rumbaugh et al.'s Object Modelling Technique (OMT) [Rumbaugh 91] and for the readers convenience a diagramming key can be found in Appendix D.

1.3. Structure of the thesis

The thesis has been divided into a further seven chapters describing key areas of research and ends with the final chapter containing the conclusions that can be drawn from the work and in particular addresses the aims expressed earlier in this chapter.

The remainder of the thesis is laid out as follows:

- Chapter 2: Knowledge within Databases

  This chapter highlights why knowledge is required within databases with respect to the three major forms of knowledge encoding i.e. structural, behavioural and explicit. It then goes on to investigate what approaches have been taken to add this knowledge, culminating in the tenet of active databases.

- Chapter 3: Review of Active Databases

  Chapter three introduces the issues relating to active databases and then goes on to survey the young but active field. The survey is structured so that each active database prototype is individually reviewed in detail and then the reviews summarised and tabulated at the end of the chapter.

- Chapter 4: The REFLEX Approach

  This chapter examines the issues involved in active database management as highlighted in chapter three, and the approaches adopted by REFLEX in their resolution.

- Chapter 5: The REFLEX Knowledge Model

  This chapter describes the EECA knowledge model of REFLEX, including its rule and event representations. These are followed by the event semantics of the model, and its event specification language called the English ESL. The condition and action specifications are also introduced with respect to their semantics.

- Chapter 6: Design Architecture and Implementation

  This chapter looks at the design decisions and implementation of the

REFLEX active database model following the semantics as described in the preceding chapters.

- Chapter 7: Evolution and Experience of REFLEX
  Chapter seven reviews the various prototypes, their findings and shortcomings. It then goes onto describe the practical interaction and use of the resultant system, followed by worked example applications.

- Chapter 8: Conclusions and Future Work
  The final chapter evaluates the work presented in this thesis and assesses whether or not it has achieved the aims expressed in this introduction. Particular consideration is given to comparing the resultant system with those surveyed in chapter three. Final comments will be addressed to pointers for future work that results from work performed for this research including using the working prototype as a tool to gather real data from active applications.

1.4. Summary

This chapter has served to introduce the research domain, that of active databases and the major goal of how best active functionality can be augmented onto existing commercial databases. Motivations for this research goal were addressed such as, the desire for an organisation to preserve its investment in its software and human resources. A number of sub-goals were highlighted such as what form should the knowledge representation take, how should the test of the internal condition be declared.

Chapter 2

Knowledge within Databases

Many authors, Mylopoulos [Mylopoulos 90] and Elmasri and Navathe [Elmasri 94], have asserted that there is a desire to move ever more domain knowledge from applications and to maintain that knowledge within databases. Since the aim of this research is to provide active knowledge management to an object-oriented database system, this chapter reviews previous approaches that have been taken to allow more domain knowledge to be maintained within the database and then introduces the tenet followed by active databases.

2.1. Introduction

During the late 1960's a major software development problem raised its head. Systems were being implemented, where the constituent applications which served an organisations' different functional units (such as Sales or Accounts) maintained their own data and file structures. As a result, major problems with data redundancy arose. In answer to this problem in 1961 the first concept of the generalized database, was envisaged by Bachman [Fry 76]. Bachman designed an Integrated Data Store for General Electric, where the data was removed from the individual application programs and stored centrally. This meant that the integrity of the data was increased i.e. it could

be relied upon as there was only ever one copy of the data (in response to the problems caused by data redundancy[1]). The concept later evolved through standardisation, e.g. the ANSI/SPARC layered model [Tsichritzis 78], to the modern day Data Base Management Systems which have the initial goal of application-data independence and further goals of multiple user views and system catalogs to store the database description (schema).

Since the focus of this research is to allow more knowledge to be represented within a database, what exactly is knowledge? Knowledge is one of those words that everyone knows the meaning of, yet finds it hard to define. Freundlich [Freundlich 90] has demonstrated that knowledge has many meanings, for example the following terms data, facts and information, are generally used synonymously with knowledge.

There is much knowledge about a domain that requires representing in a database system. Two primitive kinds of knowledge are known as *a priori* and *a posteriori* [Luger 89]. The term *a priori* is Latin for 'that which proceeds'. This sort of knowledge is independent and free from the senses. An example of *a priori* knowledge could be a statement such as 'all triangles in a plane have 180 degrees'. The opposite of *a priori* is *a posteriori* knowledge, which is derived from the senses. For example, if you saw someone with blue eyes, you would believe their eyes were blue. Later if you saw them remove blue contact lenses to reveal brown eyes, your knowledge would be revised. This chapter looks at both the *a priori* and *a posteriori* knowledge that must be encoded.

This chapter is structured as follows: Section 2.2 looks at current database systems and the knowledge that they support i.e. structural, behavioural, metadata and integrity constraints. Section 2.3 examines the Semantic Data model followed in section 2.4 by the Object Data Model. Section 2.5 introduces the active data model and finally section

---

[1]Redundancy could however still be designed in, if deemed necessary for reasons such as efficiency

2.6 summaries the chapter.

2.2. Current Database Systems

As commented on by many authors [Fry 76, Bowers 93, Elmasri 94], the database concept was contrived to achieve data independence and promote data sharing by removing the data from the application programs and storing it centrally. Hence the data, in the form of facts i.e. without meaning, was stored centrally. Knowledge as to the data's use was distributed amongst the many application program. Many authors have distinguished the differences between knowledge and data, such as Wiederhold [Wiederhold 84], where he exemplifies this distinction by means of an example citing the following assertions (i.) Mr. Lee's age is 43, data, (ii.) middle age ranges from 35 to 50, knowledge, (iii.) people of middle age are careful, knowledge and (iv.) Mr Lee has never had an accident, data. Problems with respect to knowledge redundancy were occurring, as described by Kim [Kim 95], which were analogous to the problems of data redundancy, namely inconsistency and maintainability. This scenario could be exemplified by considering the effect of modifying the underlying data-structure by the addition of a new attribute, this would cause severe maintenance problems as the many programs that use the data-structure would also need to be modified. Deductive database systems (DDS) and knowledge base systems (KBS) have both tried to allow more knowledge to be represented in their respective systems. The DDS approach has concentrated on deriving new knowledge from that which is represented explicitly [Bell 90]. Whereas the KBS approach has strived to represent knowledge declaratively, without regard to its use so that it may be shared by many applications, this could be analogous to data independence.

As domain knowledge, such as structural knowledge, is moved from the application programs into the database, new demands are placed on the database. The modelling allowed by these databases must be extended to allow richer modelling primitives,

which would allow the knowledge to be expressed correctly in a form that closely represents the real world. Shortcomings as described by Schek [Schek 91], were discovered in the relational model which is essentially record-oriented, where functional dependencies are enforced by using the concept of a key to tuples in a table, but what if the Universe of Discourse (UoD) does not map directly into tables? Hull and King cite in their survey [Hull 87], the attempts that were made to rectify this situation by developing newer data models which were progressively semantically richer. These new Semantic Data Models (SDMs) provide relationships, inheritance, objects (dynamic or behavioural properties) and integrity constraints. Traditional data models which were not afforded these rich modelling constructs turned to integrity constraints, to overcome their shortfall. As surveyed by Peckham and Maryanski [Peckham 88], in some models the integrity constraints became part of the model itself, i.e. the structural constraints. Even so, these constraints are not sufficient to model the complexity of the UoD, this is overcome by the semantics being embedded in the user programs.

The range of structural constraints were increased with the SDM which provides explicit abstract relationships, that were already provided by the Artificial Intelligence (AI) community, such as generalisation, aggregation, classification and association, as recognised by Smith and Smith [Smith 1977]. These hierarchies can themselves lead to problems, such as what is the outcome of a database update at these higher levels. For example, in the case of a student which inherits from a person superclass, if the person is updated, how will the subclass be affected? Clearly, semantics are required for such operations. These hierarchies can also be materialised by the relational databases, but the semantics of the generalisation and classification must be embedded in the user programs. This is unlike a SDM, where these relationships are provided as primitives of the system, allowing the system to maintain itself. This leads to a fundamental distinction between both approaches; in relational systems the programs can handle the hierarchy, in SDM the programs also know what to do with the hierarchy but more importantly, they know that it exists. Chakravarthy et al.

[Chakravarthy 90a] exemplify this distinction by examining integrity constraints and their use in relational systems to validate any given constraint. They take as an example the VALIDPROC procedure in DB2. Here the relational system knows how to enforce the constraint but is unaware of the constraint itself. Being hard-wired into a program the constraint cannot be used for any other purpose such as query optimisation.

Knowledge, maybe explicit or implicit. KBS strive to be make knowledge explicit. Freundlich noted in [Freundlich 90]

> "Explicit means open to direct manipulation. Within the programming context, this means removing the knowledge from the procedural setting in which it is usually embedded in conventional programming and representing it in a declarative form."

The explicit representation has many advantages namely, understandability, modularity, maintainability and extensibility. A simple data structure differs from a formal knowledge representation scheme by the possibility of being interpreted, i.e. the ability to draw inferences, allowing information to be obtained which is implicit in the knowledge base. Thus, unlike relational databases, the data available in a knowledge base system is not only the data explicitly stored but also the data that can be inferred from this knowledge. For example, if Colin is a student, the system can automatically infer that Colin is a person from the semantics of the generalisation abstraction, without it being explicitly declared. From this point of view, SDMs can be seen as rudimentary KBS where primitives are provided to represent explicitly a set of abstract relationships.

The structural features of the UoD are focused on by SDM's. For example, a *student* can be seen as a classification and a specialization of a higher abstraction of *person*, i.e. a student is a *role* that a person may take. More recently, object databases have emerged, where all information (both structural and behavioural) concerned with an object is gathered together. From the above example, a student is able to attend lectures, which defines what the student may do i.e. its behaviour. Hence, an object

is characterised by the actions it may undertake (its *interface*). The user has no interest in how the action (or *method*) is performed, simply how it is invoked by the sending of *messages*.

There are many SDMs of which SDM [Hammer 81], TAXIS [Mylopoulos 90] and IFO [Abiteboul 87] are well-known examples. They are sometimes referred to as structurally object oriented models [Dittrich 86] since they are characterised by their structural, relational and attributive features.

The following sections describe the prominent models, semantic and object with respect to their encoding of domain knowledge, and then goes on to investigate the different kinds of knowledge, with a view as to how they may be represented.

2.3. Semantic Data Model

Semantic data models provide a high level of abstraction for modelling data. This is analogous to the trend in programming languages where low-level languages evolved to ALGOL-like languages which were able to provide richer, more convenient programming abstractions; which according to Hull and King [Hull 87], buffer the user from low-level machine considerations. This allows the data to be modelled more akin to the real world. Semantic data models were primarily introduced as schema design aids, but are increasingly being developed into full-fledged database management systems. Semantic data models attempt to explicitly capture a rich set of relations among real world entities.

As described by Nierstrasz [Nierstrasz 89], the major abstractions for modelling this real world knowledge provided with the SDMs are *classification* where a collection of entities or objects are considered or taxonomised as a higher level class; *generalisation* which allows a higher level class to be synthesised from many similar objects. Its

inverse is *specialisation* where classes are further refined into more specialized classes. The *aggregation* concept allows composite objects to be constructed from component objects. These abstractions as pointed out by Hull and King [Hull 87] allow more semantics to be represented explicitly. As stated earlier, the system itself can maintain abstractions such as generalization and can thus, remove the burden of maintaining these structural abstractions from the user to the database.

There are some SDMs which have addressed the dynamic aspects of the UoD, for example TAXIS [Mylopoulos 86], SHM+ [Brodie 84] and *the event model* [King 84]. TAXIS manipulates its transactions, exceptions and exception handlers as detached entities, which results in their ability to be arranged in hierarchies and have attributes. The transactions are described in terms of the entities involved, i.e. its parameters, the type constraints on the participant entities and the set of sub actions that comprise the definition of the transaction. Since the transactions are parameterised by the entities involved, transactions can be specialised along with the entities.

This section examined SDMs in terms of the knowledge the represent, further reviews of the SDMs can be found in the literature. Hull and King [Hull 87] present various models with respect to a common example. Whereas Peckham and Maryanski [Peckham 88] compare the SDMs and their support of relationships, the abstractions they represent, and their approach (if any exists) to dynamic modelling. Albano [Albano 89] presents a comparison of advanced SDMs, such as TAXIS.

The following section discusses object-oriented databases and how real world domain knowledge is represented within them.

2.4. Object Data Model

Since the target host for this research is an object-oriented (OO) database, this section

considers the types of knowledge that OO systems encode. Object-orientation is said to model concepts from the real world in a direct and natural manner, similar to SDMs. It accomplishes this by modelling an object in terms of its structural entity, its related knowledge of being i.e. its behavioural characteristics, and the events that trigger operations that change the state of the objects.

In the case of systems modelling, Mylopoulos [Mylopoulos 90] states that a notation can be said to be object oriented, "*when it encourages a direct and natural correspondence between components of notation instances and objects of application*". Following from this statement the relational data model cannot be considered to be object oriented since an entity in the process of normalisation can be split between different tables.

Even though the paradigm is becoming widespread, there is not a common understanding of what an object is. Programming languages, design methodologies, user interfaces, databases, and operating systems have all been described as being object-oriented. Even though it appears that object-orientation is common to all of these diverse areas, it soon becomes evident that the same term is being used in different ways in each domain. The Laguana Beech Experiment [Stonebraker 89a] exemplified this, as a group of leading database researchers found that there was little common understanding of the term even between themselves.

2.4.1. Object-Oriented Databases

A promising way forward is that of object-oriented database management systems (OODBMSs). Being OO they encourage a direct mapping between concepts in the real world and their computer representation, embodying both the structural and behavioural features of the UoD. OODBMS provide features required by newer applications, such as: *richer data modelling constructs* since conventional relational

systems cannot support complex data types (such as arrays, objects, classes) and *inter-object reference* i.e. more implicit knowledge. This would prevent the flattening of objects so that they fit the data model; *long transactions* as opposed to the short transactions for conventional database systems which assume that transactions last for only a short duration and thus lock very little data. The transactions for the new applications are much longer in length and thus a new form of locking is required; *version support*; *performance* since relational databases are value based, and thus are very expensive, in terms of time. In order to retrieve a required record, the values of the attributes must be searched for before the record can be retrieved. Modern applications require almost immediate response i.e. a fetch object in a CAD package.

It can be difficult to find a set of characteristics that can be held for any database that claims to be OO because there is a lack of formal definition. This is further exasperated by systems which claim to have object features but have different development paths. For instance there are systems that have been built by enhancing OO programming languages e.g. Gemstone [Copeland 84] and ADAM [Paton 89], relational DBs e.g. POSTGRES [Stonebraker 90], from semantic data models e.g. SIM [Jagannathan 88]. Not all OODBMSs found in the literature share exactly the same features. For the purposes of this research an OODBMS is deemed to have standard DBMS functions, as described by Zdonik and Maier [Zdonik 90], i.e. persistence, transactions and object features i.e. abstraction, object identity and hierarchies.

A prime feature that distinguishes an object database from a relational database is in its concept of an Object Identifier and will be discussed in the following section.

2.4.1.1. Object Identifier

An object has a system defined surrogate number as its identifier. This object identifier (OID), is used to reference the object. Identification has been addressed independently by both programming languages and databases. In the former, the object is identified

by memory reference (or by user defined labels to the memory locations). Khoshafian [Khoshafian 86] has identified that this mixes the concept of addressability (i.e. how to access an object in a given environment) and identity that is internal to the object and which should be independent of how it is accessed. Conversely, conventional database systems such as the relational systems reference tuples by the values of their attributes, identified by key or primary attributes. For example, a personnel relation may have the tuples keyed or referenced on the name and telephone number of a person. But if a person changes their telephone number, it is more difficult to locate the record. Additionally, if the person changes their name (by deed-poll for instance), the record is even more difficult to locate as the key has changed. Khosafian cites this as a major problem for referential integrity of relational systems and causing constraints to be placed such as: the primary attributes are not permitted to change even though they are descriptive properties of the object; extra primary attributes have to be used even though they are not required, for example, if the name and age are required for a person, one should not need to include the National Insurance number simply for the sake of providing a unique identifier.

Thus with both (programming languages with memory references and relational databases using primary attributes or values) of the above approaches, identity is mixed with addressability. Object identifiers are excellent for promoting referential integrity as a given object always has the same OID, regardless of the values of its attributes.

Object identifiers enhance the efficiency of a system by providing logical pointers to the required objects, and thereby avoiding expensive join operations. The pointers in object oriented databases and the pointers in hierarchical databases are similar except that the pointers in hierarchal databases are physical. The object identifier is not reusable or modifiable, hence it is impossible to change the value of the surrogate number or when an object is deleted its surrogate cannot be reused by a new object.

2.4.1.2. Impedance Mismatch

Object-oriented databases provide rich modelling features of the UoD, and also help solve the *impedance mismatch* problem, as described by Copeland and Maier [Copeland 84]. This metaphor originates from the field of electrical engineering, and refers to the fact that an impedance mismatch in an electrical circuit will prevent the maximum possible power transfer from being achieved. Zdonik and Maier [Zdonik 90] have commented that their are two aspects of this impedance mismatch: i. conventional programming languages (COBOL, Pascal, C) and DML query languages (SQL, QUEL) differ in terms of the descriptions of their data structures. The type systems of most programming languages do not support the relational structures directly, thus requiring complex mappings. Such mappings lead to a loss of information at the interface of programming language and database, similar to the case with electrical circuits. Another consideration is that since programming languages do not understand database structures, type correctness cannot be checked for ii. Programming languages are procedural whilst query languages are declarative in nature. The units of data transfer between the database and the program are smaller than the collection relations, leading to much inefficiency. This leads to unnatural and complex programming.

For example, suppose a database exists consisting of an EMPLOYEE and a DEPARTMENT table. In the program, one may be tempted to layout structures that will hold rows retrieved from each table:

```
struct employee {                    struct dept {
        char    name[20];                    char    name[10];
        date    birthdate;                   struct  employee*
        struct  dept*  department;                           depthead;
};                                   };
```

This scheme could produce a number of problems. First, the C++ structures represent the connections between employees and departments using pointers, while the database

system (if it is relational) will handle these connections via foreign keys, which will be stored as strings. Next the employee structure includes a member of type date, which could be a class for which the user may have built methods that allow the program to easily perform sorting or compression operations on calendar dates. If the internal storage format of the date as handled by the database is different, further conversion functions which transfer data from one format to the other need to be built. With an object-oriented system, this mismatch simply does not occur because the representation in the database and in working memory is identical. All referenced objects are also loaded, with the pointers properly 'wired' or *swizzled*[2] between the two representations.

A survey of the concepts of object-oriented technology can be found in [Nierstrasz 89, Stonebraker 90] and many others. For this reason, the basic fundamentals will be assumed as known. The following section introduces the approach followed in active databases.

2.5. Active Databases

The systems of today which utilise database technology, may not suffer from the problems of the original systems of the 1960's, i.e. data redundancy, they do however, have similar problems. For example consider a scenario where a new system to manage a '*Supply and Distribution Warehouse*' is developed. It is based on a central database, and has a number of application programs, each for a different sub-system, e.g. sales, accounts. Each application program would access the database and expect a certain data structure. If however, the data structure or system schema was amended e.g. an attribute was changed, then a major maintenance task would have to be undertaken to resolve the problem of redundancy within the application programs, with the possible

[2]Swizzling is a term used by Carey [Carey 91] which refers to the process of moving an in-memory object to and from its disk representation.

attendant problems this may cause, e.g. replication of knowledge, effort and possible inconsistencies.

Even though data independence is a central tenet to database theory, where the data held in the database is managed independently of any application program that utilises it, this still does not mean that the data is truly independent. Is the data model or schema really stable? If in the example of the above paragraph where an existing attribute of a table is changed or even deleted, the application programs that use that particular table and which expect the attribute to be of a certain form, will have to undergo maintenance amendments. Does this mean that the application is independent of the data? Clearly, the logical data model is not as independent as would be desired.

Active databases which attempt to resolve the problem of *application logic redundancy*[3], have been defined by Dittrich and Dayal as:

> "a database is said to be an active database if it supports the storage and maintenance of domain knowledge (or general application logic) alongside the data, and the knowledge is triggered (or activated) on the raising of events" [Dittrich 91].

The systems encapsulate an enterprise's domain knowledge within the database. Thus providing a Data and Application Logic Base System. The domain knowledge is *centralized* in one place, i.e. within the database management system itself, as opposed to being scattered across many application programs as discussed above. This approach attempts to resolve the problems of data independence since if the data model is amended, any application program changes are simply made to the logic within the database and not to the many application programs which contain the replicated access code. The domain knowledge or application logic may be represented in many forms,

---

[3]Logic required to perform certain tasks is embedded within all application programs that require the task to be performed, providing maintenance problems.

but in most active prototype systems the general form is that of modified *production rules*.

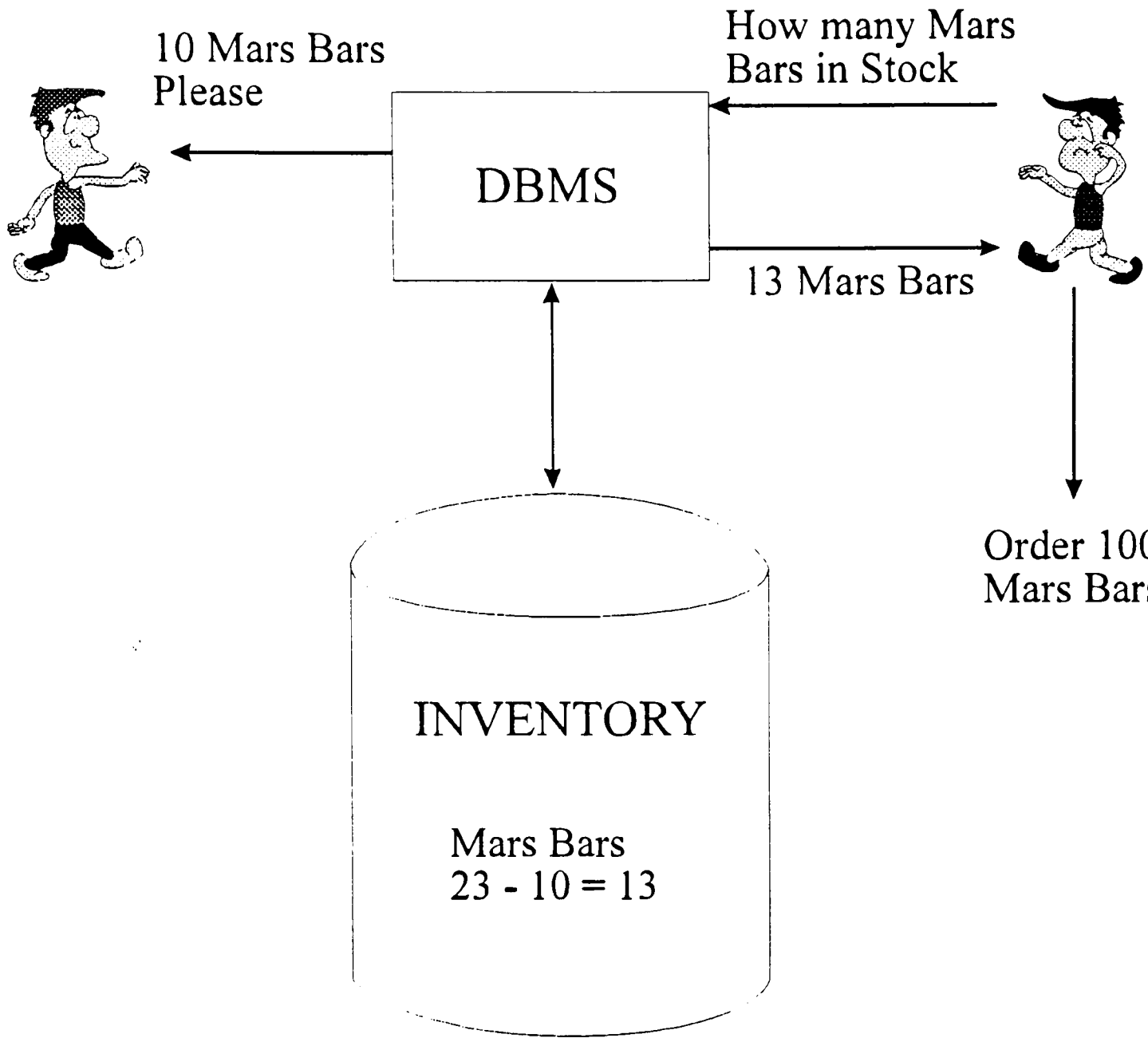


Figure 2.1 Passive Database System

Another major problem that database solutions do not presently address, is that of *timeliness* of data. This is best illustrated by means of an example, again we will use the warehouse scenario. If a customer purchases an item, it ultimately leaves the warehouse, and hence the number of items on hand is decreased by the number of items sold or distributed. At the end of the working day (or week or polling period),

an application program is run against the database which will evaluate which items require reordering, and will place them in a reorder request list. This case is illustrated in figure 2.1 with an example where a customer purchases 10 confectionary bars, after a period a query is run against the database to determine if the quantity-on-hand is below any reorder level, if so then the reorder quantity (1000 in this case) is reordered. This means that during the wait for the system to check which items are not currently stocked, the item may not have been reordered in time and hence caused a loss of business. This timeliness, for a village retailer may not be important, but for a Currency Trading System, where every split second is worth millions of pounds, could be critical. The interval between polling periods may, however, be reduced causing the database to be polled more frequently. This approach however, causes the database to test its state continually rather than carry out its intended application even in todays technologically advanced world since the major bottleneck is I/O rather than processor bound, for which parallel technologies could improve the situation. Instead of continually polling the database, another popular approach is to add logic, in the application code which updates the database, to test if any specific state has been reached. Maintenance for the overall database system becomes problematic as there is duplication of code to test the semantic integrity of the data amongst the many application programs. The redundant code has to be maintained which leads to excess cost.

The timeliness of data, is maintained in the active database by the use of the *event paradigm*. A database is said to become active (or is woken-up), on certain events being raised unlike traditional or passive databases which only perform actions when explicitly requested to do so, either by the user or by application programs. This is illustrated by the example in figure 2.2, where on the customer purchasing the confectionary bars, the database is activated and its knowledge is triggered. The database then processes the logic and places the reorder request automatically and most importantly, on time.

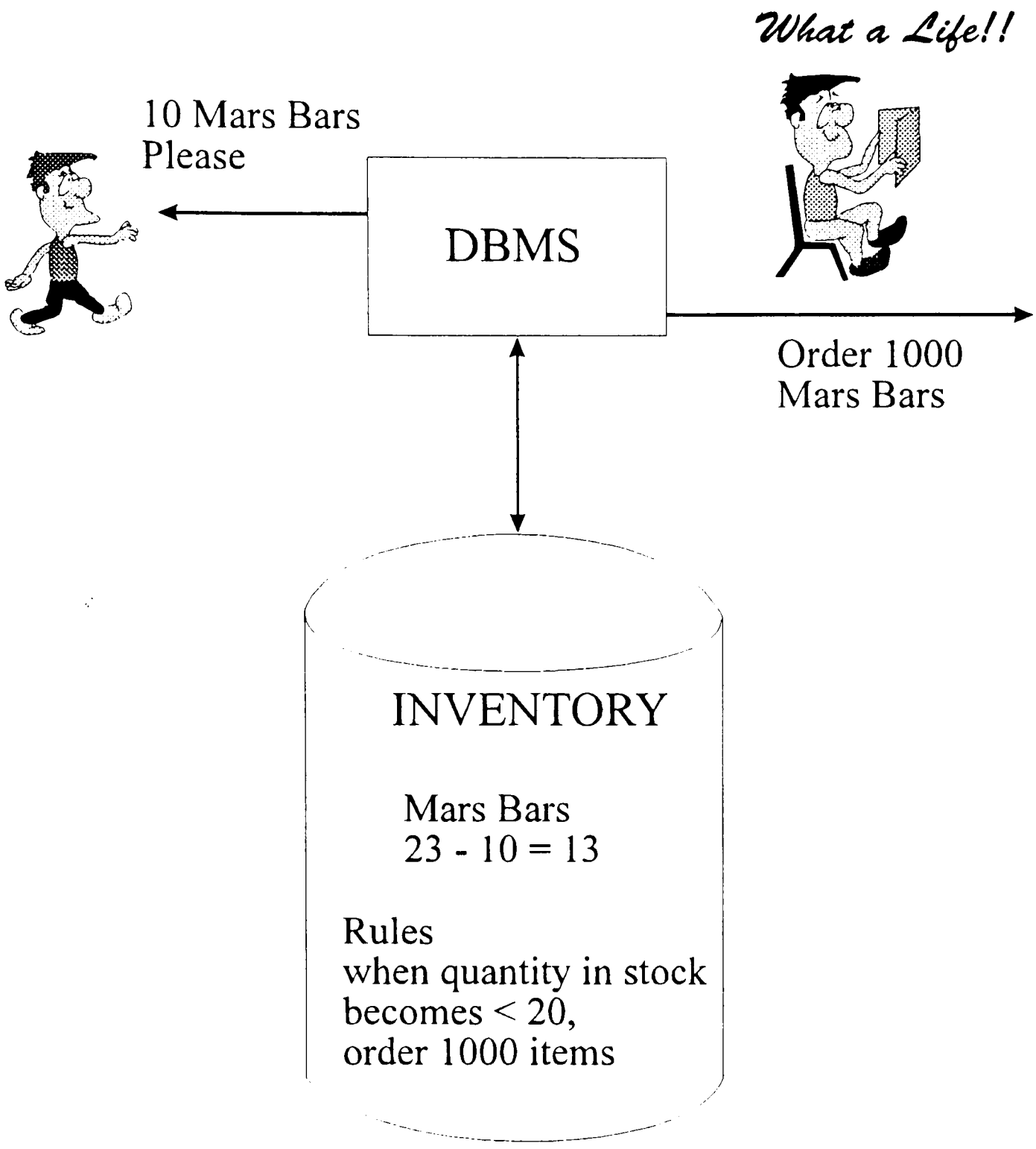


Figure 2.2 Active Database System

Active databases maintain knowledge which is triggered on the occurrence of events, this knowledge is generally structured using the Event-Condition-Action (ECA) [Dayal 88, Dayal 89] knowledge model, which is composed of a production rule tuple of the *antecedent-consequent* type. If the antecedents or left-hand side of the rule is satisfied then the consequents or right-hand side of the rule will be actioned. The production rule must also take into consideration the specification of the triggering event(s).

McCarthy and Dayal have proposed the format of an ECA rule [McCarthy 89]:

Event	ON	event-clause
Condition	IF	condition-clause holds
Action	THEN	execute action

Not only does the condition or a database state or condition have to be ascertained but also an event has to be raised first. The event-clause/condition-clause combination are collectively known as the '*situation*' and the THEN (or action) part is sometimes known as the '*reaction*' [Dittrich 91]. The situation and reactions must be specified.

2.6. Summary

This chapter provided motivation for trying to encode knowledge with a database system and then reviewed the methods currently used to encode the intensional UoD as opposed to the extension. A discussion of the distinction between knowledge and data was made, which concurred with the views expressed by Freundlich [Freundlich 90], such as "Knowledge can be embodied in a program as a procedure or as a data structure. This distinction corresponds to the philosophical difference between knowing *how* to and knowing *that*".

Current database systems were discussed with respect to the desire to encode more domain knowledge within the database systems. Following, newer models were progressively introduced initially, the Semantic Data Model (SDM) which is semantically richer providing features such as generalisation, aggregation and association and thus allowed greater facilities to encode more intentional knowledge. The Object Data Model followed, which like the SDM provides rich semantic modelling i.e. classification, generalisation, aggregation and association, but also but promotes behavioural modelling and hence affords features such as reusability and

extensibility. These models are relevant to the research because they illustrate how intensional knowledge is represented within the schemas. However, the object data model is important because it is the underlying data model for this research's active data model and prototype.

The field of active databases were later introduced illustrating the two main problems that they attempt to resolve which were, i) the problems associated with application logic redundancy and ii) the timeliness of the data. The ECA knowledge model was overviewed. The various forms of encoding different types of knowledge were discussed throughout the chapter, illustrating the varied research aims being pursued.

The following chapter introduces issues for active databases and a literature survey.

Chapter 3

Review of Active Databases

This chapter introduces the issues within active database systems research and discusses the pertinent design issues involved. A survey of current research in the area is presented which examines how the issues are tackled by the different research prototypes.

3.1. Introduction

Active databases, the domain of this research, was introduced in the previous chapter together with other forms of knowledge representation systems. This chapter introduces the issues concerning active database research. However, before the issues can be discussed, a view of a canonical active database architecture may prove useful. According to McCarthy and Dayal [McCarthy 89] an active database must manage knowledge, generally in the form of productions rules, and respond to the occurrence of any specified events. In order to execute this task active databases have some form of the following components: a rule or knowledge manager, an event detector, a rule evaluator, a condition evaluator and an execution module.

The remainder of the chapter is structured as follows, the issues concerning active databases are introduced such as, whether the underlying technology affects the feasibility of such a system and events and their representation. Section 3.3. provides a survey into

the current state of the art active databases. It is followed by a comparison of the various models, and an evaluation of their features.

3.2. Issues of Active Databases

This section serves to highlight some of the issues and raises some questions regarding active database theory. The questions are open, and are initially introduced and then only later examined in the literature review section, by observing how the related active database prototypes attempt to provide solutions. The solutions provided by this research to the questions below and further questions are examined in chapter four which overview this research's active database model called **REFLEX**, and then chapters five and six discuss the knowledge model and the active models design and implementation, respectively.

3.2.1 Underlying Architecture

An initial premise for this research was to extend a database with active functionality, with a concern being whether the underlying architecture affected the feasibility of the active database system. In answer to this concern, it was ascertained that the ability to support activity was unrelated to the underlying architecture (e.g. relational, hierarchical, network or object-oriented), i.e. not affected by the technology. Since, activity or automatic application defined reactions on predefined triggering events is not the exclusive domain of any one database technology. To put this into perspective, the old CODASYL network data model of 1972, had procedures definable for entities. The CODASYL data model had the keyword **ON**, which was followed by a database operation or an error trap. If the event occurs, the procedure is called. It was not however, sophisticated enough to evaluate a condition as well as an event.

Newer databases such as IBM's Starburst [Lohman 91] & University of California's POSTGRES [Stonebraker 87] are both based on the relational theory. Rules and their extensions have been added to both the above systems and have proved to be successful. The rules, in the case of Starburst, act upon whole relations in one operation.

Research efforts such as HiPAC [Chakravarthy 89] and ADAM [Paton 89], provide active extensions to object-oriented database theory. Object-oriented databases seem to encompass rule extensions with greater ease than the other older technologies, such as the traditional relational model. This may be because they have more semantic facilities such as classification, inheritance and encapsulation, which allow additional functionality to be added to higher order classes. Alternatively, perhaps this may be because of their relative youth since they are not restricted to a certain data model or that they serve a large user base. Since object-oriented databases are still research prototypes[1] they can thus tackle the new theories as they emerge.

The above illustrated that the concerns raised were unfounded i.e. the underlying technology would not affect the feasibility of an active database system, as they have been constructed on various technologies. The later literature review highlights the underlying technology of each database.

A fundamental component for active database systems is the event. The issues concerning events will be introduced in the following section.

---

[1]Even commercial offerings, such as ONTOS, are still essentially used in research laboratories and not in mainstream applications.

3.2.2. Events

An event is a happening or occurrence of something of interest and hence must be detected in order to activate the database. Once the event is detected, if it affects a rule it may bring the rule into context so that its condition clause may be tested.

Detectable events have been categorised [Chakravarthy 89], in three broad groups:

- internal to the database
  these could be updates, reads on the database; or transaction points such as the start of a transaction or its committal. These are generally equivalent to the data manipulation language (DML) commands available i.e. UPDATE, SELECT.

- temporally based
  events based on clock e.g. at specific points in time, relative or periodic. To allow the detection of temporal events, a clock input to the Event Manager, provides the triggering event. Examples of temporal events could be *absolute* at 5pm, *periodic* every 5 minutes, or *relative* after 5 hours.

- externally defined by user applications.
  these are events which are external to the host database system and are either user or application defined. Examples of such events are those raised by a radar detecting an aircraft within its airspace, and are detected by the application program making an event raise call to the Event Manager.

Events which may cause a rule to be brought into context could be *primitive* i.e. a single atomic event, or *complex* i.e. where a number of primitive events are allowed, joined together using a logical algebra e.g. conjunction, disjunction, etc. Simple or primitive

events are relatively easy to understand. They are said to occur *instantaneously*, at a specific point in time, unlike conditions which *hold* over certain intervals or periods of time. Complex events blur the definition of an event because they are composed of many primitive events combined in an algebra (English ESL in the case of REFLEX), and hence do not occur in an instance but over an interval[2], similar to conditions except that conditions relate to database states i.e. the values of data objects; whilst, with respect to active databases, events may[3] or may not do so.

This research categorises complex events into two groups, homogeneous or heterogeneous. The ability to support heterogeneous events affords considerable flexibility and power over the support of homogeneous events alone, and thus can be used to determine the intended use of a given research prototype.

Homogeneous events can be defined, in the case of this research, as

> "*a complex event which is composed of primitive events of the same category i.e. internal, temporal or external*".

Example homogeneous events:

(a) UPDATE PERSON AND UPDATE STUDENT

(b) ON DATE 16/3/95 OR ON DATE 30/10/93

Similarly heterogeneous events can be defined as

> "*a complex event which is composed of primitive events which span the various categories i.e. internal, temporal or external*".

---

[2]A complex event occurs at the point of occurrence of the last valid primitive event. This is described later in the chapter five, section on Event Specification.

[3]In some systems, events can be seen as conditions. For example, this is the case with logic and especially temporal logics [Kowalski 86, Knight 88].

Example heterogeneous events

(a) UPDATE STUDENT AND DAY IS SUNDAY

(b) EVENT RADAR-PULSE AND UPDATE AIRCRAFT

The literature review will look at the different active database research prototypes and how the above issues are tackled i.e. whether the database allows primitive or complex homogeneous/heterogeneous events, and also the following such as: how long after the occurrence of an event can the event still be used in the evaluation of a rule's event specification clause, i.e. is the event valid. If it is used against a rule's specification, is it still available for a different rule's event specification clause. The number of rules the event (or events) affects or brings into context within the different research prototypes i.e. a single rule, or many, is examined.

3.2.3. Analysis and Design of Rules

The extracting of rules from an enterprise and the subsequent design of the rules in the database, requires careful attention. In addition to traditional database design, *Activity Design* also takes place where the business rules of a domain are extracted and the rules are designed for the domain. This latter area is more difficult than the former. This is because, each rule may cause a change of database state, and since the rules may inter-relate, each fired rule causes further changes of state, i.e. the database may continually generate events and on actioning the events generate further events. Thus the cyclic process may go on forever and not allow the database to stop.

A typical example could be the following, where on making a change to a students record, its status is checked which forces a change in a table, which in turn forces a change in the primary table.

```
Rule 1      ON      UPDATE STUDENT
            IF      select name
                    from STUDENT
                    where gradeAverage < 30;
            THEN    update STUDENTUNIT profile="FAIL"

Rule 2      ON      UPDATE STUDENTUNIT
            IF      select name
                    from STUDENTUNIT
                    where profile = "FAIL";
            THEN    update STUDENT profile="FAIL"
```

The above example illustrates a situation where on the STUDENT table being updated, Rule 1 is triggered. This then performs an update on the STUDENTUNIT table, which triggers Rule 2, and vice-versa. Hence, the database will continuously serve the two rules cyclically forever i.e. the rules will not terminate.

3.2.4. Rule Termination

The firing of a rule may then lead to subsequent firing of further rules, which may trigger themselves indefinitely i.e. infinite loops. This may prove disastrous for a database system for example, control could be lost between sets of interacting rules, rules could fill both main memory or disk by continually performing inserts on a table, causing the system to crash. At best a disaster could be nothing more than the system simply slowing down, as a result of serving its rule invocations. This situation must be avoided or at least controlled, but how can the system be brought back under control? In answer to this concern, a number of strategies exist. The design of rules should be examined to ensure that no cyclic interactions are possible, Aiken et al. [Aiken 92] propose application of static analysis algorithms. These algorithms may be used to provide information about three properties of rule behaviour to a database rule programmer. The properties are:

i. Termination

Can the termination of rule processing be guaranteed after a change in database state?

ii. Confluence

Similar to the law of commutation where the order of the execution of rules may or may not affect the final resultant state of the database. For example, if multiple rules are triggered, does the final database state depend on which is executed first? If it does not, the rule set is said to be confluent.

iii. Observable Determinism

If the action of a rule is visible to the environment i.e. it may perform a rollback or modify some data, then it is said to be observable. Similar to confluence, if the order of execution of non-prioritized rules does not cause a change in the order observable actions, the rule set is said to be observably deterministic.

A more common approach that is adopted by many systems, is to monitor the run-time invocations to prevent infinite loops by counting the rule executions and comparing against a pre-defined system limit. A further approach is to detect the occurrence of the same rule again but with the same set of activators i.e. given situation.

3.2.5. Transactions and Coupling States

Multiuser and multiprocess database systems can operate concurrently because they support the concept of transactions. A transaction is an atomic unit of processing, which is performed in its entirety or not at all [Bell 92, Gray 93]. To facilitate transaction management and specifically recovery management (where a transaction fails, recover to the previous state), the following operations need to be tracked:

- Transaction Start

  Marks the beginning of transaction execution

- Transaction Commit
  Signals the *successful end* of a transaction so that any changes executed within the transaction can be safely committed to the database.

- Transaction Abort
  Signals the transaction has *ended unsuccessfully*, so that any changes applied within the transaction must be undone.

In active databases, by their very nature, processes[4] are interrupted by the raising of events and the possible invocation of knowledge processing. These interruptions, themselves self-contained transactions, can be declared to occur relative to the interrupted transaction, by the specification of *coupling modes*. Coupling modes, originated in the HiPAC project [Chakravarthy 89], as described by Dayal [Dayal 89] define how events, conditions and actions relate to the database transactions. Coupling modes allow the designer to specify whether a rule's conditions or actions should execute in the triggering transaction or a separate transaction. These coupling modes are not available in other active database prototypes i.e. Starburst [Lohman 91] or POSTGRES [Stonebraker 91b], where the rules conditions and actions are executed in the same transaction as the triggering event, and hence are not as flexible.

For an ECA rule the coupling anchors available to a transaction are the Event-Condition (E-C) and the Condition-Action (C-A). In the former, the coupling modes of immediate, deferred or decoupled are offered to the evaluation of the condition on an event being raised. For example, if a process is executing against a database, figure 3.1, and an event occurs, if the event affects a rule the rule's event specification must first be evaluated (assuming the occurring event has a higher priority than the executing process). If the event specification of a rule is satisfied i.e. the event raised causes a rule to be brought into context, then the condition clause of the rule must be evaluated. The rules designer can determine whether the evaluation of the condition clause is to be performed with

---

[4]For simplicity, a uni-processor architecture machine is assumed.

respect to the interrupted transaction in one of three modes (i.) immediately and control returned to the original process after the evaluation has completed, figure 3.1 (a), or (ii.) whether the evaluation be deferred until the original process has completed, figure 3.1 (b) or (iii.) whether the two processes be decoupled and performed in parallel, figure 3.1 (c).

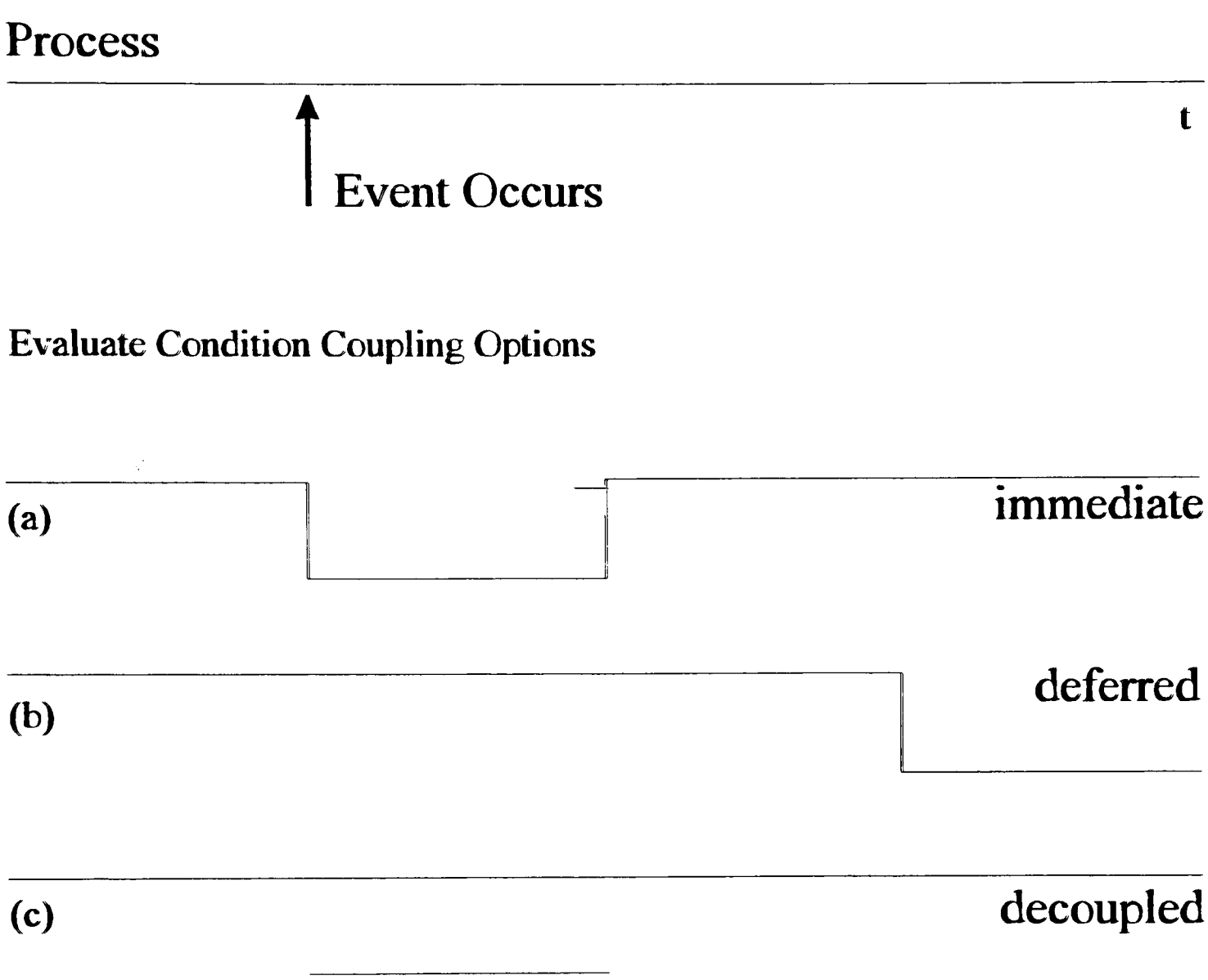


Figure 3.1 **Coupling Modes**

For the case of the Condition-Action coupling, again the modes of immediate, deferred and decoupled are offered, for the execution of the action clause with respect to the execution of the condition.

Splitting the coupling modes into the two anchor types either E-C or C-A causes extra problems as the number of permutations increases. For example, if coupling modes of immediate/deferred are offered on an event, the condition clause relative to the parent transaction will be immediately evaluated and if it is satisfied, the original operation is

continued until it is at point of committal then the deferred action clause is executed. Obviously a contradictory but perfectly valid situation, since why test the state of the database immediately but then defer any action. For example, from the Air Traffic Control scenario, if the movement of an aircraft is detected by the Radar, the system interrupts its current task and evaluates whether the aircraft is in danger of collision, if so, the system continues its prior interrupted task, and when completed it then takes the deferred action to prevent a disaster.

Interrelationships between the primary or host operation and triggering transactions may exist for example, what if a triggered (host) transaction is at a point of committal, and the deferred (triggering) action fails, does the primary operation abort or commit? The same problem would be cited in a case of decoupled/decoupled transactions. Where both on an event, the condition of the rule is evaluated in a separated spawned process and if it is satisfied, the action clause is also executed in a separate new transaction. In these cases a causality constraint or some sort of dependency between the host and interrupting transaction may be supported, to indicate what will happen in the case above i.e. the interrupting transaction may only commit if the host transaction commits. For example, if the user is entering data about a particular aircraft, this may cause an interrupting rule to be fired where the action is decoupled which inserts information into a log. If the user then aborts the data entry for some reason, i.e. the wrong aircraft number was used, should the decoupled entry to the log also abort.

3.2.6. Rule Contention

If many rules are triggered by the same event, they are said to be brought into context. A rule whose situation (patterns of both event specification and condition clause) is satisfied is said to be activated or instantiated. Multiple activated rules may be on the agenda at the same time. In this case, the inference engine must, generally, select one rule for firing. This selection may be based on a number of alternative strategies. The rules may be fired in order of retrieval, or based on priority. Another approach may allow the rules to execute

concurrently. At this point active databases are very different to conventional memory based knowledgebases or expert systems since the rules may have coupling modes.

The selection strategy is made more complex when the issue of coupling modes is considered. If rules have different coupling modes, the priority assigned to a rule, should take into account the urgency of situation evaluation. For example, lower priority rules should not be afforded an immediate coupling mode since this would cause a conflict, since higher priority rules would be evaluated first.

For how long after the detection of the situation is a rule able to fire. An apt analogy for this scenario may be considered as in neurophysiology, the study of the nervous system. Where an individual cell or neuron emits an electrical signal when stimulated. No amount of further stimulation can cause the neuron to fire again for a short time period. This phenomenon was reported by Brownston et al. in their work on OPS5 and is called *refraction* [Brownston 85]. That is, if the same rule kept firing on the same fact over and over again, the system would never accomplish any useful work. The refraction of a system is generally kept to a minimum. i.e. a rule only fires once given a situation occurring. This may however be left to the rule designers discretion.

3.2.7. Knowledge Coupling

As well as transactions which have coupling modes between triggered and triggering transactions, the degree of coupling between an active database and its underlying data model, is important since it is a measure of the portability and adaptability of an active data model. This measure allows the determination of whether the active features of a model can be applied to different data models, or whether they are restricted to a single database. As the literature review will illustrate, most prototypes are tightly coupled to their underlying data model.

3.2.8. Knowledge Representation

Since active databases attempt to encode domain knowledge within a database system, two primitives of this knowledge i.e. rules and events, must be represented. There are many representation strategies that may be followed, they may be

i. Hard-Wired

The rules may be hard-wired into the application system code, as in Ode [Gehani 92a]. This is advantageous for the application programmer, since the rules may be coded in. This however, has disadvantages such as, the declaration of rules requires a application language programmer and the rules must be declared prior to compile time. This means they cannot be modified or added to without re-compilation.

ii. Metadata

This is the general method for representing rules in relational system such as POSTGRES [Stonebraker 91b], Starburst [Lohman 91], Ariel [Hanson 92] and now in commercial offerings. Rules are defined as metadata in the schema, together with tables, integrity constraints, view. Operations are provided to add, drop or modify rules.

iii. First-Class Objects

In object-oriented environments, rules may be represented as first-class objects, as with HiPAC [Chakravarthy 89] and ADAM [Diaz 91b]. This means that the rules are instances of a rule class, and hence like other objects they can have attributes and can be subject to the standard database manipulation and security features.

The following section provides a survey of the current state of the art active database systems, and will investigate the knowledge representation mechanisms employed as well as previously mentioned issues.

3.3. Literature Review

Active databases are a current popular area of research. As such, there is much work in the area. In this section the *state-of-the-art* active databases are reviewed, by considering the following framework:

- underlying model
- their knowledge model
- support for existing applications
- support for new non-traditional applications
- what makes it novel

Particular emphasis is placed on the knowledge models of respective active databases, since this is a major area of interest in this research. After the major salient features of the alternative active databases have been discussed, the differences are highlighted in *table 3.2*.

3.3.1. POSTGRES

POSTGRES [Stonebraker 87], a progression from relational INGRES, started its development life in 1986, at the University of California. Stonebraker and Kemnitz [Stonebraker 91b] report that the motivation for the project was the recognition that the next-generation applications required two further dimensions from the original dimension of data, those of *object management* and *knowledge management*. Hence, POSTGRES, an *extended relational* system, attempts to add the concepts of object abstraction and closer coupling between the knowledge base and a relational DBMS.

One of the prime aims of this review is to concentrate on the knowledge model that POSTGRES promotes and not cover the details, these are readily available [Stonebraker 89b, Stonebraker 91b], except where the details are deemed necessary for the prime aim.

POSTGRES has increased structural knowledge by the provision of *classes* (or relations/types), which may inherit from other types, which provides some degree of semantic richness. However, the inclusion of methods i.e. functions internal to an object as found in object-oriented/class based systems, in the database are not allowed. This is because it is language neutral i.e. it is not bound to a particular programming language, and so cannot allow methods to be attached without becoming biased towards a programming language. It does however, provide three different kinds of functions: C functions, operators and POSTQUEL functions.

In addition to POSTGRES's four major constructs i.e. *classes*; *inheritance*; *types* and *functions*, it also provides knowledge management by means of two *rules* systems. These will be reviewed in the following section.

3.3.1.1. Rule System

As stated by Stonebraker and Kemnitz [Stonebraker 91b], the design of the POSTGRES rules system was governed by the desire to construct one general purpose rules system, which would be able to perform all of the following: *view management*, *triggers*, *integrity constraints*, *referential integrity*, *protection* and *version control*. This aim is at odds with other systems such as Starburst [Lohman 91], where the creation of views is handled by the extension of the query language using Starburst's eXtended Normal Form (XFN). The view is defined and stored in the data dictionary, i.e. views are not covered by the rules system and different structures are required for different functionality.

There are two implementations of the POSTGRES rules system. One through record level processing which is a part of the run-time system. This is called when individual records are accessed etc. The second implementation is through a *query rewrite* module. This

code exists between the query optimizer and the parser. It converts a user command to an alternate form prior to optimization. POSTGRES does not however, provide an automatic rule method chooser, so the user must decide on which is the best method of rule system for a given rule.

The record level rules system is implemented as an extension to POSTGRES's query language POSTQUEL. An extra clause, the ON clause, has been added which allows the triggering event to be declared.

The rule system has the following syntax:

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

From the above query, the first line is the event declaration clause. The triggering event is related to an object. The events POSTGRES can detect are illustrated in table 3.1.

Event Types	**Events**
Internal	retrieve, replace, delete, append new (i.e. replace or append) or old (i.e. delete or replace)
Temporal	time () date () functions
External	not supported

Table 3.1 POSTGRES detectable events

The object referenced in the clause is the name of a class or class column (attribute). The optional keyword INSTEAD indicates that the POSTQUEL commands are to be executed instead of the action which caused the rule to activate. If the keyword INSTEAD is not present, then both the action and user event are executed. The POSTQUEL commands for the rules system, are the same as the normal POSTQUEL commands but with two additional changes:

i. the keywords *new* or *current* can appear instead of the name of the class preceding any attribute.

ii. refuse (target-list) is added as a new POSTQUEL command.

Rules may additionally specify actions to be taken as a result of user updates. As can be observed from the valid events listed above, POSTGRES allows events to be retrievals as well as updates.

3.3.1.2. Summary

POSTGRES being a post-INGRES system, does try to provide a superset of facilities provided by INGRES, i.e. support for inheritance, and abstract data types. It has not however succeeded in this goal since basic query operators, such as union, intersection and other set functions have not been implemented. This fact restricts the applications that can be implemented on POSTGRES and hence, it has essentially been used by academic institutions as an research/exploration tool for future database requirements, i.e. object management and rule management.

In terms of its knowledge management facilities, it allows more structural knowledge to be encoded within the data structures, similar to Sematic Data Models as surveyed by Hull and King [Hull 87], such as classification and aggregation facilities to compose complex objects. For explicit knowledge representation, it provides two implementations of rules systems which may be seen as complimentary, i.e. one is tuple or record based, the other a crude form of set-processing which is realized by a converter module which sits between the parser and query optimizer. It does not however, have any conflict resolution strategies except simply that rules are fired in sequential order of occurrence.

The successor to POSTGRES is being specified and designed. It has, imaginatively, been designated POSTGRES II [Stonebraker 91a].

3.3.2. STARBURST

The Starburst project [Lohman 91] at the IBM Almaden Research Centre in San Jose, California was initiated in 1985 to redress the problems faced by conventional database management systems. Its goal was to build from scratch an *extensible* DBMS prototype that would both

i. allow the DBMS to have the functionality to serve the new application requirements efficiently
ii. to provide a test-bed for IBM's own ongoing research in DBMS technology.

The impetus for the Starburst project arose during the early 1980's when a version of System R was adapted to create a *distributed* relational DBMS prototype, called R* [Lindsay 80]. This did not prove successful and Lohman et al. [Lohman 91] reported the following:

> "*The lesson was clear: extensibility cannot be retrofitted; it must be a fundamental goal and permeate every aspect of the design*".

The research team, by basing Starburst on the relational model and on extensions of a *standard* database access language, could exploit much of the proven relational DBMS technology and its theoretical foundations. It also facilitated porting existing applications to Starburst. Starburst was designed with a *common relational data model* with domain-specific extensions, as new areas are researched.

For the purposes of this survey, only the active database extensions made to Starburst will be considered.

In Starburst's approach to active extensions, user defined rules respond to aggregate or cumulative changes to the database. This, according to the Starburst research team, matches more closely the set-oriented paradigm of relational systems and leads to cleaner more natural semantics, because typically many rules may be triggered at any given point

[Lohman 91]. Other systems such as HiPAC [Chakravarthy 89] and POSTGRES [Stonebraker 91b] differ in that their rules respond to operations on a single row i.e. a single record (although POSTGRES does support a minimal rules system which is set-oriented).

Starburst is made active by two rule systems: a relationally oriented production rule system and an object-oriented system, called Alert, that monitors objects and invocation of methods. Both are described in turn.

3.3.2.1. Production Rules

As other rules systems, Starburst's rules have trigger, condition and action clauses. The *trigger clause* may specify one of the SQL operations INSERT, DELETE or UPDATE as events on the *trigger table*, identified by the keyword ON. The rule's *condition clause*, signified by the IF keyword, is any SQL query. If the query is satisfied, Starburst executes the *action clause*, which is any sequence of database commands, preceded by the keyword THEN. Actions may suppress changes to the database by terminating the current transaction or perform further updates which may trigger further rules to fire. A user may temporarily DE-ACTIVATE defined rules and RE-ACTIVATE them later. The rules may refer to transition tables which contain changes made to the tables since the beginning of the transaction.

An example rule to support referential integrity between Department and the Employee tables, where each table has the DeptNo attribute, could be:

```
CREATE RULE delete_department
ON Department
WHEN DELETED
IF      'SELECT *
        FROM Employee
        WHERE DeptNo IN
```

```
                (SELECT DeptNo
                        FROM deletedDepartment AS (DELETED()))',
    THEN 'ROLLBACK WORK';
```

The above rule would rollback (or abort the transaction), where a department was deleted which still has employees attached to it.

The rule processor is invoked at transaction completion. Rules may also contain PRECEDES and FOLLOWS clauses to specify a partial order for rule processing. Starburst's production rules are fully integrated with Starburst. And hence, the rules are stored in the system catalogs as metadata.

3.3.2.2. Alert

Starburst also has another method of encoding rules within the database system, called Alert. This method differs from the Startburst Production rules system in that even though both systems are based on SQL, as reported by Schreier et al. [Schreier 91] the production rules system can only refer to events that refer to built in operations: update, insert and delete, whereas Alert rules may monitor user-defined operations, like *pay* on views. Hence, Alert rules are at a higher level of abstraction than with the production rules system. Unlike the production system, the Alert rules must be *explicitly activated* [Lohman 91].

The Alert system is based on SQL views, where queries (termed active queries) are conducted over active tables (which are append-only views). The Alert rule is declared using the CREATE RULE statement (which may be read as create view), followed by the SELECT clause contains the rule's actions (which may be user-defined functions) and the FROM and WHERE clauses express the rule's condition.

An example Alert rule could be:

```
        CREATE RULE      user1_condition AS
```

```
SELECT      empName, expenseAmount
FROM        activeTable_Journal
WHERE       methodDescription='expenseClaim'
            and expenseAmount > 2000
```

Whenever the methodDescription is called, the rule is activated, and the rule fired if the expense amount is greater than 2000.

3.3.2.3. Summary

Both of Starburst rule systems support temporary tables which are only available during the current transaction. The rule triggers are deferred until the end of the transaction commit time.

In terms of support for existing applications, even though Starburst is an extended relational system, it is only a development prototyping system, that may one day produce a future DBMS or at least define its features. Hence, it does not really try to support the existing applications but to investigate what facilities are required by the new applications.

3.3.3. HiPAC

The HiPAC (**Hi**gh **P**erformance **AC**tive database management system) research project [Chakravarthy 89] began its development in 1986 at the Xerox AIT, although its underlying PROBE object-oriented data model began its life in 1984 [Manola 86]. Since HiPAC is an object-oriented DBMS, the rules in HiPAC, as all other forms of data, are treated as objects. There is a rule object class, and every rule is an instance of this class.

The project originally addressed two critical problems in time constrained data management: handling of *time constraints* and avoidance of wasteful polling i.e. *active database management*. These goals were further augmented by a goal of *contingency plans* i.e.

alternate actions that can be invoked whenever the system determines that it cannot complete a task in time.

HiPAC has developed knowledge and execution models. The knowledge model provides primitives for defining timing constraints, situation-action rules. The execution model allows various coupling modes between transactions, situation-evaluations and actions. These are examined in detail below.

3.3.3.1. Knowledge Model

The primary objective for HiPAC was to develop a knowledge model that provides primitives for defining situation-action rules.

The HiPAC knowledge model is built on the PROBE data model [Manola 86]. In PROBE, the real-world objects are modeled as *entities*. The *attributes*, *relationships* and *operations* are modeled as *functions*. The necessary extensions for HiPAC are: *rule objects*, specific *temporal constructs* for expressing events, and execution model primitives. The rules themselves are modeled as first class objects i.e. they are instances of a rule class.

The HiPAC project [Dayal 88] in its knowledge model originated the ***Event-Condition-Action (ECA)*** rules. These ECA rules have been used as the basis for many other active database systems i.e. Starburst [Lohman 91, Schreier 91], Ode [Gehani 92a], Adam [Diaz 91b]. HiPAC also introduced coupling modes, which specify when the condition (EC) or action (CA) is evaluated relative to the transaction, and supports *immediate, separate,* and *deferred* modes. HiPAC, supports complex events (i.e. collection of primitive events) as triggers for its ECA rules. It also, unlike the Starburst production rules system, allows rule actions to be defined by the application (external events), and allows rule actions to contain requests to applications i.e. applications to define and signal their own events.

The condition clause is a collection of object-oriented DML (Data Manipulation Language) query.

The execution model consists of a nested transaction model, and sub-transactions for condition evaluation and action execution, and parent transaction based on coupling modes.

3.3.3.2. Architecture

HiPAC was implemented on an object-oriented database with nested transactions. The object database being the PROBE data model. PROBE is intrinsic to HiPAC. The Knowledge model was implemented as part of transaction manager. The transaction manager noted the triggering event for the rule. When a rule is created, the situation part of the rule is passed to the condition monitor. The execution model is executed in the transaction manager. The underlying data model had to support the semantics of rule object class including detecting events, determining which rules to fire on events, scheduling condition evaluation and action execution according to coupling modes.

3.3.3.3. Summary

In summary, the HiPAC database research project, in its attempt to find solutions to problems of the handling of time constraints and active database management, contributed both the ECA model, and the EC/CA coupling modes.

It should be noted that HiPAC is an *in-main-memory* database. Hence it does not have the same problems of real large disc-based database systems which have to access terabytes of data. It can use technology that is available in the Expert System domain, such as the Rete match algorithm [Forgy 82]. This situation was asserted by Dittrich and Dayal [Dittrich 91], who reported that disk based active database systems cannot take advantage of these AI solutions since they were not designed for the large database

domain and do not scale up. Instead the use of query optimization techniques are used for the recognise part of the recognise-act cycle of AI systems.

3.3.4. ADAM

ADAM (**A**berdeen **DA**ta **M**odel) [Paton 89] is an object-oriented database implemented in PROLOG, to which rule processing has been added [Diaz 91a, 91b]. Being implemented in Prolog, frames [Minsky 75] were chosen as the rule representation method, but Paton and Diaz [Paton 91] assert that the frames were extended to objects by the enforcing of encapsulation and addition of methods. Within frame systems, as described by Kingston et al. in their work of CRL a frame system [Kingston 87], demons are used to represent both behaviour and derived values where event-triggered demons can be invoked on the update of a frame. In contrast methods in object-oriented systems are called explicitly.

3.3.4.1. The Knowledge Model

As stated by Diaz et al. [Diaz 91b], in terms of providing rule processing, "The focus is on providing a uniform approach". Hence ADAM models all components of the knowledge model uniformly as objects, including rules and events.

The structure of a rule is mainly described by the event that triggers the rule, the condition to be checked and the action to be performed [Diaz 91]. A rule can only specify a single, simple event in its event specification clause. Hence the relationship between an event and a rule is 1:M, that is, an event may affect many rules, but a rule may only be triggered by a single primitive event.

The rules, being modeled as first-class objects, have familiar attributes and methods required for their E-C-A description. They also have two further attributes *is-it-enabled* and *disabled-for* which specify the status of the rule. The attribute is-it-enabled describes

the status at the level of the whole class appearing as the active-class value, whereas the disabled-for attribute describes the status for specific instances of the class.

An example ADAM rule could be as follows, where an integrity constraint that maintains that students are below the age of seventy:

```
new ([OID, [
        event([3@db_event]),
        active_class([student]),
        is_it_enabled([true]),
        disabled_for([1@student, 23@student]),
        condition ([(
                current_arguments([StudentAge]),
                        StudentAge > 70
        )]),
        action ([(
                current_object(TheStudent),
                current_arguments([StudentAge]),
                get_cname(StudentName) => TheStudent,
                writeln (['The student ', StudentName,
                                'with age ', StudentAge,
                                'exceeds the expected age']),
                fail
        )])
]) => integrity_rule.
```

ADAM as described by Paton and Diaz [Paton 90] supports metaclasses, which allow the run-time creation of classes. Hence objects are considered to be metaclasses, classes or instances. When the system is compiled, the metaclass called *meta-class* already exists. All subsequent classes are created by sending messages to metaclasses i.e. *meta-class*, such as *new* for a new class, *put_slot* and *put_method* which create the new attributes and methods respectfully.

3.3.4.2. Summary

ADAM's rule processing facility is influenced by the HiPAC research project, where active facility is implemented upon an object-oriented database using ECA rules. ADAM does however benefit from being implemented in an interpreted Prolog environment, the major benefit being extensibility. Since the environment allows the creation of new classes, objects at run-time.

ADAM is limited in that its rules may only have one primitive event specified against them similar to Starburst and POSTGRES. The events may be generated from a number of generators, some of which may be in external applications as in HiPAC, but the events must be based in methods of the application classes. ADAM does not address the issues of rule contention, optimization or transactions.

3.3.5. ODE

Ode [Agrawal 89] is an object-oriented database system, developed at AT&T Bell Laboratories. The database is defined, queried and manipulated using the database programming language O++, which is an extension to the object-oriented programming language C++ [Stroustrap 86].

The constraint and trigger mechanisms in Ode make it an active database [Gehani 91]. Even though providing integrity constraint facilities is not a new issue, Ode provides facilities for object-oriented databases that can be used to specify complex and higher-level integrity constraints. The purpose of constraints is to ensure data consistency while that of triggers is to perform actions when some conditions are satisfied.

Ode supports two kinds of constraints: hard and soft. Hard constraints are checked after each object access while soft constraints are checked just prior to a transaction commit. Three kinds of triggers are supported: once-only, timed and perpetual. Triggers, unlike constraints, must be activated explicitly.

Constraints and triggers have been implemented independently since they are conceptually and semantically different.

3.3.5.1. Event-Action (EA) Model

Unlike most other active database system, which use the ECA model, Ode has proposed an Event-Action (EA) [Gehani 92a] model. The EA model allows the condition clause to be *folded* into the event specification. This has the advantage of reducing the number of coupling modes between the event and action (the complexity of the condition clause coupling modes have been eliminated). It does however, limit the functionality of the overall system for a number of reasons.

The first and most obvious disadvantage is that in order to test the event clause, which includes the condition statement (i.e. a mask), the evaluation of the clause is sought with undue inefficiency. This is caused by the evaluation of conditional statements even if they were not brought into context by the triggering event i.e. the event specification alone was not satisfied. The result of the event clause is not known until the conditional part of the specification is also tested. This can be exemplified by the following example:

$$\text{UPDATE student}_{e1} \text{ AND UPDATE profile}_{e2} \text{ AND (Student.name = "Fred")}_{mask}$$

If either of the events e1 or e2 occur, the above rule clause will be tested. The rule cannot fire however, until the entire clause is satisfied. What happens if one of the events never occurs? The following scenarios may take place, either

- the condition mask is not evaluated until the final event occurs. But then the entire event part of the clause must be satisfied before the condition mask may be tested. This is no different from the conventional ECA model where the condition is a separate clause, which may only be tested once the event clause has been satisfied. Their approach simply removes the possibility of an EC coupling mode, other than the implicit immediate.

- the condition mask is evaluated before all of the required events have occurred, in this example assuming that one of events may never occur, but for what gain. This will be inefficient as the result may never be used, and in fact the side-effects of evaluating a non-requisite query could be unknown.

The rationale given for the EA model, as described by Gehani et al. [Gehani 92b], with its combined event and condition clause, is that ODE is essentially a programming environment and the EA model with its less complicated coupling modes facilitates the programming goals of efficiency and optimisation.

Another subtle disadvantage of the EA model, which is common with other active models, is the inability to handle *external* condition clauses. If the condition part of the clause is based on the state of the external environment i.e. readings from a thermometer, rather than that of the internal state of the database, this may be difficult to extract from the integrated event and condition clause that Ode proposes. For Ode to handle the condition based on the external environment, dummy updates are required to the database in order for the internal condition evaluation to take place, i.e. an application program will read the thermometer and update a thermometer table, which may cause any rules on the table to be tested.

To complement the reduction of the ECA model to the EA model, Gehani et al. [Gehani 92a] have also illustrated a further coupling mode in addition to immediate, deferred, separated (decoupled). The fourth mode is in effect an expansion of the separate mode and is broken into two as follows: separate dependent and separate independent.

The EA rules are specified within the program code for the objects to which they apply. The format for a trigger could be using the following template:

```
class name {
```

```
        ...
        trigger:
                trigger-list
        ...
};
```

Where the trigger-list is a list of triggers each of which is specified as

trigger-name(parameters): [perpetual] event ==> trigger-action

The trigger must be explicitly activated by calling its name as in method invocation, if the keyword perpetual is used, the trigger remains available until explicitly deactivated.

An example Ode rule to enforce the constraint that students must be under seventy could be:

```
class student {
        ...
        int StudentAge
        ...
trigger:
        T1():perpetual before create(i) && i.StudentAge > 70
                ==> tabort;
};
```

Ode has the ability to recognise complex events, for which Gehani et al. have proposed an event specification language [Gehani 92a, 92b]. The language provides the primitives for the combination of events using the logical combinations i.e. conjunction and disjunction.

3.3.5.2. Summary

Ode is an attempt to provide a default persistent store to C++, as reported by Agrawal and Gehani [Agrawal 89]. Later this goal was extended to support active behaviour [Gehani 92a]. Attempting to extended a programming language with persistence and

activity meant that many of the goals were based at the programming language i.e. optimisation and efficiency and not the normal database goals of integrity and flexibility, as acknowledged by Gehani [Gehani 92b]. Because of the desire to simply the programmability, the ECA model was reduced to the EA model. Ode does however provide an event specification language, which although specified using finite automata does lack in semantics of operation, as highlighted by Widom [Widom 93].

3.3.6. Event/Trigger Mechanism (ETM)

At the University of Karlsruhe, the Event/Trigger Mechanism [Dittrich 86, Kotz 88] was designed to enforce complexity constraints in design databases (DDBS). The ETM was motivated from ideas derived from exception handling in programming languages.

The goal of the project was to enforce consistency constraints by triggering checking at arbitrary times and to execute user-definable reactions to consistency violations. This is similar to exception handling mechanisms from programming languages and interrupt mechanisms from hardware.

The ETM has several parts i.e. consistency constraints, events, actions and triggers. The consistency constraints were explicitly inserted into the database, and then explicitly checked using a CHECK [constraint name] call and finally deleted using the REMOVE keyword. Events are system attributes and are defined using the EVENT keyword. This will assign the event a unique system-wide identifier. Events are generally raised explicitly, because as stated by Dittrich et al. [Dittrich 86] "this approach is feasible as we can assume that most of the knowledge on what event should be meaningfully raised, at what time, rests with the user or with the application program (and frequently nowhere else)". Actions are host language or DML statements. Triggers are the mechanism for pairing the event to the action and have the following format:

TRIGGER <trigger name> =
ON <event name>

DO <action name>

After a trigger has been defined it must be explicitly activated, and later deactivated.

ETM was later implemented on top of Damascus, a prototype design database system. Damascus is a development database system, upon which to test ideas i.e. prototyping. Hence the system source code was available to amend. The resultant functionality of the system was limited in terms of its transaction coupling mode to that of immediate only.

3.3.6.1. Summary

The ETM provided the fundamentals for active database management, even though the majority of all rule invocation was explicit. It has two complementary concepts, those of consistency constraints and triggers. The consistency constraints had to be checked explicitly. The triggers, although they could be triggered by a single event, the event itself had to be explicitly raised.

3.4. Comparison of Approaches

Only HiPAC, Ode and ETM support external events. The need for temporal events was recognised by HiPAC and Ode which both proposed absolute and relative events. POSTGRES, on the other hand, supports a few specific temporal events (e.g. time and date). POSTGRES and Starburst support only disjunction of events whereas HiPAC provides three event constructors: disjunction, sequence and closure allowing a regular event expression to be expressed.

Starburst supports only the deferred coupling mode, while POSTGRES, ADAM, ETM, Sybase [Sybase 90] and InterBase [Interbase 90] support immediate coupling mode only. HiPAC supports a general execution model [Hsu 88] which includes immediate, deferred and detached modes. The detached mode includes *causally-dependent* and *causally-independent* modes. In causally-dependent mode there is a commit dependency between

the triggering transaction and the rules triggered by that transaction. All allow cascaded execution of rules. Again, all of the systems, except Sybase, support multiple rules to be associated with a relation. Sybase allows only three rules per relation, one each for INSERT, DELETE, and MODIFY events. All of the systems, except HiPAC, prioritize potentially executable rules activated by an event. ETM, POSTGRES and InterBase order rules in the order specified by the user when rules are defined. HiPAC interleaves multiple rule execution (i.e. provide concurrent rule execution) using an extended nested transaction model, even so it allows the serialization order to be specified. Starburst assumes a conflict resolution scheme similar to the ones used in expert systems. Starburst uses an incrementally computed context for execution of rules which were triggered by an event.

Table 3.2 Features of Current Active Database Systems

Features/Database	StarBurst	POSTGRES	HiPAC	ADAM	ODE	ETM
Status	Prototype	Prototype	Prototype	Prototype	Prototype	Prototype
Underlying Technology **Real Database**	Extended Relational Yes	Extended Relational Yes	Object-Oriented No, Memory	Object-Oriented Yes	Object-Oriented Yes	File System No, Memory
Degree of coupling to Underlying Model	Tight	Tight	Tight	Tight	Tight	Tight
Architecture **Extensibility** **Flexibility**	High High	Low High	High High	High High	High Low	Low Low
Data Model	Ext. Relational Object	Ext. Relational Object	Object-Oriented	Object-Oriented	Object-Oriented	Design
Knowledge Model **Rule unit** **Rules as 1st Class Objects** **Events as 1st Class Objects** **Primitive/Complex Events** **No of Events** **External Events** **External Conditions**	Production Rules Set-oriented No No only Disjunction 1 No No	Production Rules Tuple No No only Disjunction 1 No No	Production Rules Object Yes No Complex >1 Yes No	Frames Object Yes Yes Primitive 1 Based in Methods No	Triggers & Constraints Object, Set at-a-time No No Complex >1 Yes No	Triggers & Constraints Triggers No No Primitive 1 No No
Execution Model **Conflict Resolution** **Termination of Rules** **Optimization** **Transactions** **Coupling Modes** **Number of Actions**	Partial Order No - Nested Deferred 1	Sequential on Priority No Yes Nested Early/Late 1	Concurrent Execution No Yes Nested Immediate, Deferred, Decoupled 1	No No No Nested Immediate 1	No No No Nested Immediate, Deferred, Decoupled 1	No No No Nested Immediate Compound Statement
User Interface **End User** **Developer**	SQL C++	POSTQUEL C++	N/A- Common LISP/Flavours	N/A Prolog	OdeView O++	N/A C

3.5. Summary

Newer applications require timely responses, otherwise their information becomes out-of-date. The traditional passive databases could not furnish the time requirements, without causing unmanageable redundancy of code or undesired polling of the database. Active databases fill this niche. Their ability to allow the database to hold both the data and the knowledge required by an enterprise leads to elegant handling of both.

This chapter introduced the issues concerning active databases. It then went on to survey *state-of-the-art* active database systems and discovered that there remains a void that requires attention i.e. more powerful facilities are required such as the ability to specify many actions for a given situation, increasing the expressiveness of rules (i.e. more complex triggering event specification language) and further efficiencies to be gained by distribution and parallelism.

The following chapter addresses these issues and this research forwards the REFLEX Active Data Model as a solution. It attempts to be a more comprehensive data model, but still remain portable and adaptive.

Chapter 4

The REFLEX Approach

The previous chapter introduced the important issues concerning active database technology and raised some questions. It went on to review several prominent research prototype active database systems, with a view as to how they addressed the earlier questions. This chapter examines the issues raised and provides considerations and justifications for the approach taken within the active database model forwarded by this research.

4.1. Introduction

The main objective of this research is to investigate how *best to augment an existing database management system with active functionality*, in order to preserve legacy systems and the investment therein. With this in mind the major aims are that the resultant system should be portable, adaptive, flexible and efficient, i.e. the system should be available on more than one platform and that it should accommodate or adapt to new databases so that its additional functionality is transparent to the host database. The active database model introduced by this research is called REFLEX. It was so named since one of its design goals, as described in chapter one, is to enable a host database to respond to a given situation *reflexively*.

This chapter addresses some the issues introduced in the previous chapter, and explains how the REFLEX active data model differs from related work. The chapter proceeds as follows, section 4.2 introduces the underlying technology used by REFLEX. This is followed by a section on the loose coupling model that REFLEX introduces to allow it to adapt to new underlying host databases. Section 4.4 and 4.5. describe the knowledge and execution models respectively. REFLEXs self-active features are discussed in Section 4.6, and knowledge integrity in section 4.7. Finally, section 4.8 summaries the chapter.

4.2. Underlying Technology

Since the answer to the issue as to whether the underlying technology would affect an active databases feasibility, was that it does not, the next question for this research was what underlying technology to use. Related research like Starburst [Lohman 91] or POSTGRES [Stonebraker 87] both attempt to extend the domain of relational technology. Whereas the HiPAC [Chakravarthy 89] and ADAM [Paton 89] systems provides activity for object-oriented databases.

Having an aim of being portable, REFLEX should in theory be implementable on any given platform and underlying technology (i.e. relational or object-oriented), but this research limits the scope to a single technology. The portability between platforms is examined by multiple implementations, and discussed later in chapter six. The choice of object-oriented databases as the underlying technology was made because of the inherent reuse of base classes i.e. additional active functionality may be added by specializing a base class into an active subclass. This will be discussed in greater depth in chapter six. Further motivation is that object technology may be the next evolutionary step for relational systems as highlighted by many authors, such as Schek and Scholl [Schek 91], and Kim [Kim93].

4.3. Loose Coupling

As a major tenet arising from the design goals to be both adaptive and portable, the system should be *loosely-coupled* to the underlying database and model. By loose coupling it is meant that the active extension is added to the host database via a defined interface layer, figure 4.1. The active extension is not given access to the internal code of the host, but must call services as required. This approach is unlike other active database prototypes described in the literature i.e. POSTGRES, Starburst, HiPAC, Ode, ADAM, which are tightly coupled or entwined to their underlying database management system code. Hence REFLEX is loosely-coupled since the active knowledge extension is a distinct layer on top of a given host database management system, allowing it to be '*bolted-on*' to a DBMS.

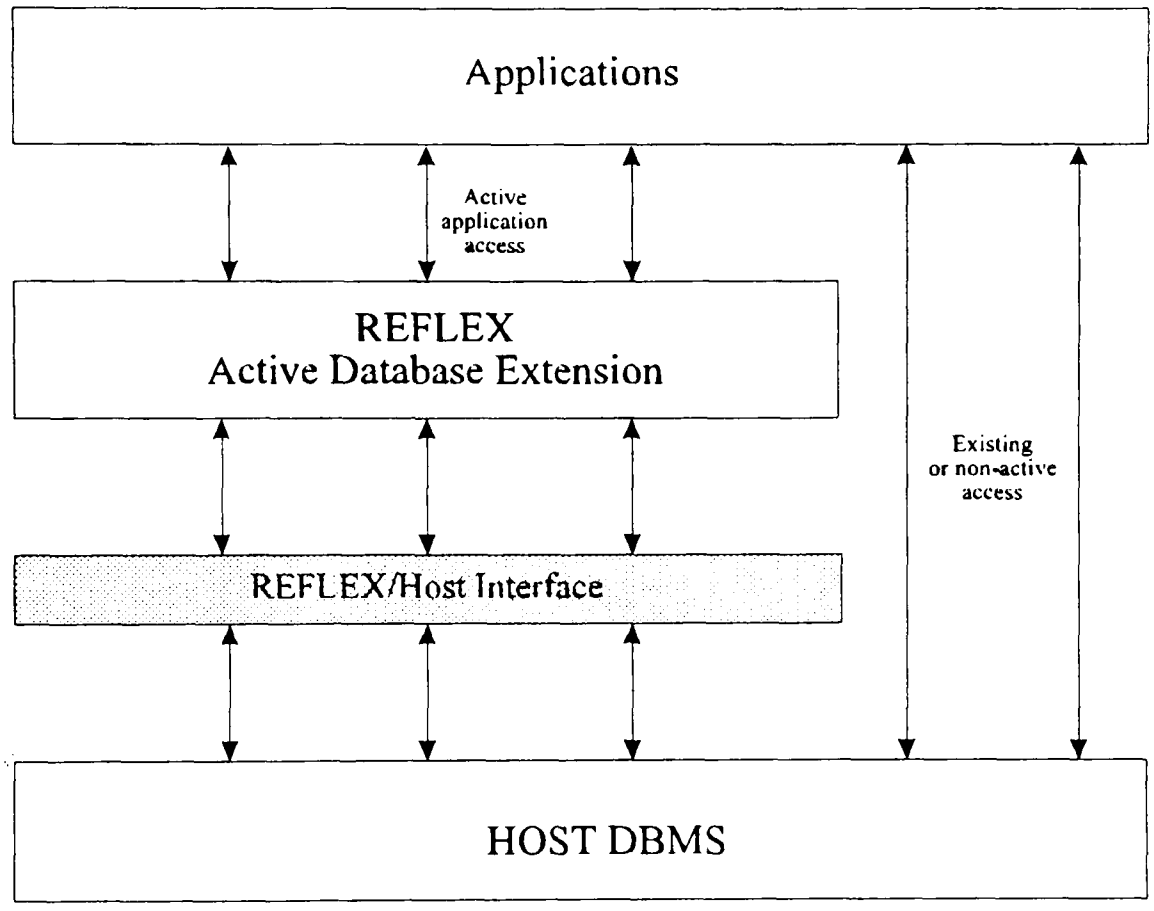


Figure 4.1 Layered access to the host DBMS

This loose-coupling is achieved by having a code *wedge* (like those found in interrupt service routines, ISRs), which is inserted between an application and the DBMS. It intercepts calls to the DBMS, and invokes some of its own processing logic before

allowing the call to go through. If the call has no significance to REFLEX, then the call is allowed through, unhindered. The module that performs this task of actually making the physical contact with the underlying host database system is called the Transparent Interface Manager (TIM), and is discussed later in chapter 6. Another analogy could be that TIM is very similar to a *gateway* as described by Brodie [Brodie 93], where access to a resource is routed via a filtering layer.

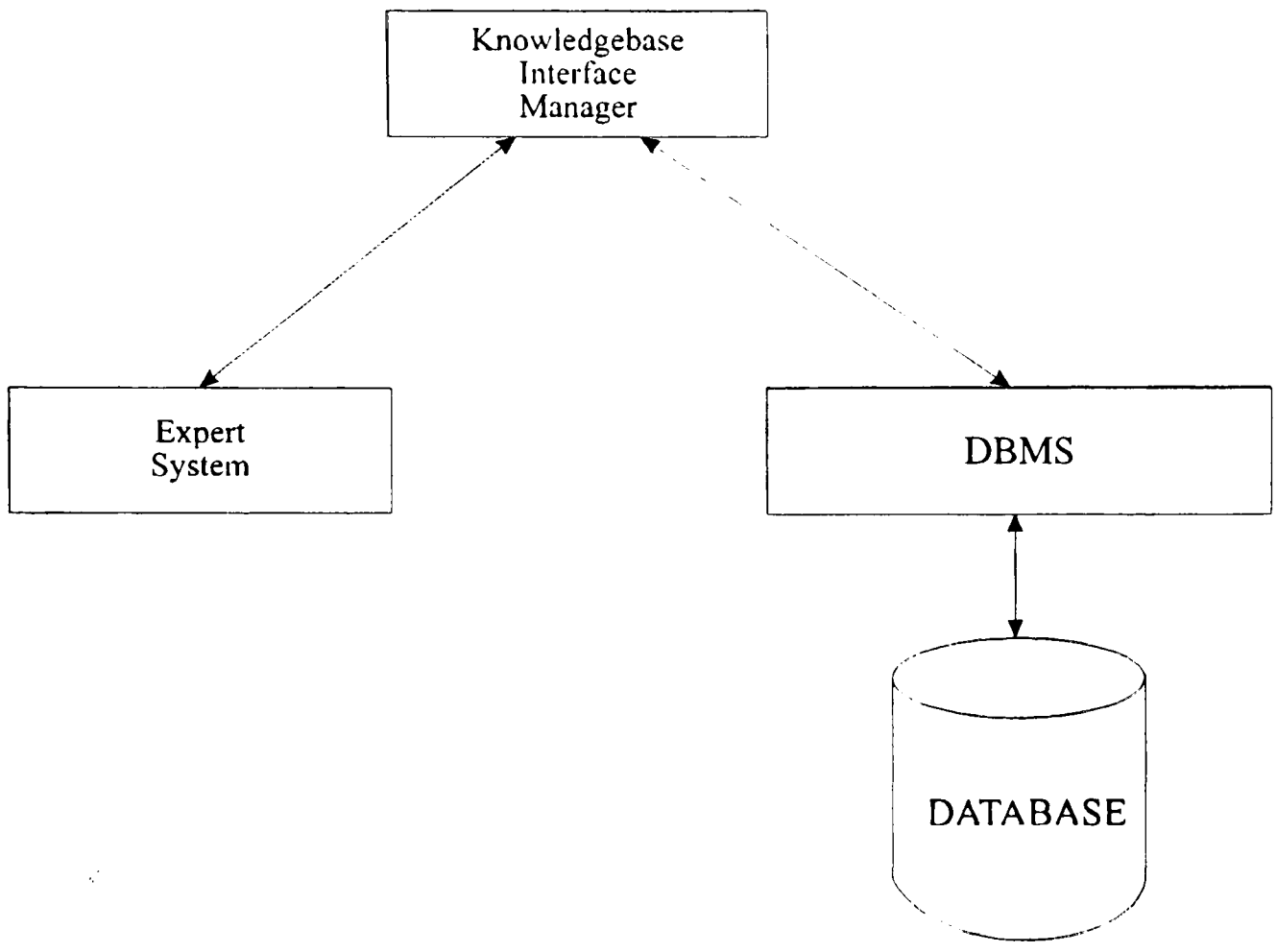


Figure 4.2 Knowledgebase system approach

The approach taken by REFLEX is unlike that of knowledge base systems (KBS), figure 4.2., where the component parts are distinct and consist of an expert system and a database coupled together by a third part, the common data channel [Beynon-Davis 91]. For KBS the communication between the expert system and the database is via

messages of some kind routed or administered by the knowledgebase manager. Knowledge-based systems traditionally assume that the data needed resides in main memory. The KBS approach means that the knowledge-data coupling is weak or loose since the knowledge is held and maintained by the expert system and the data is held

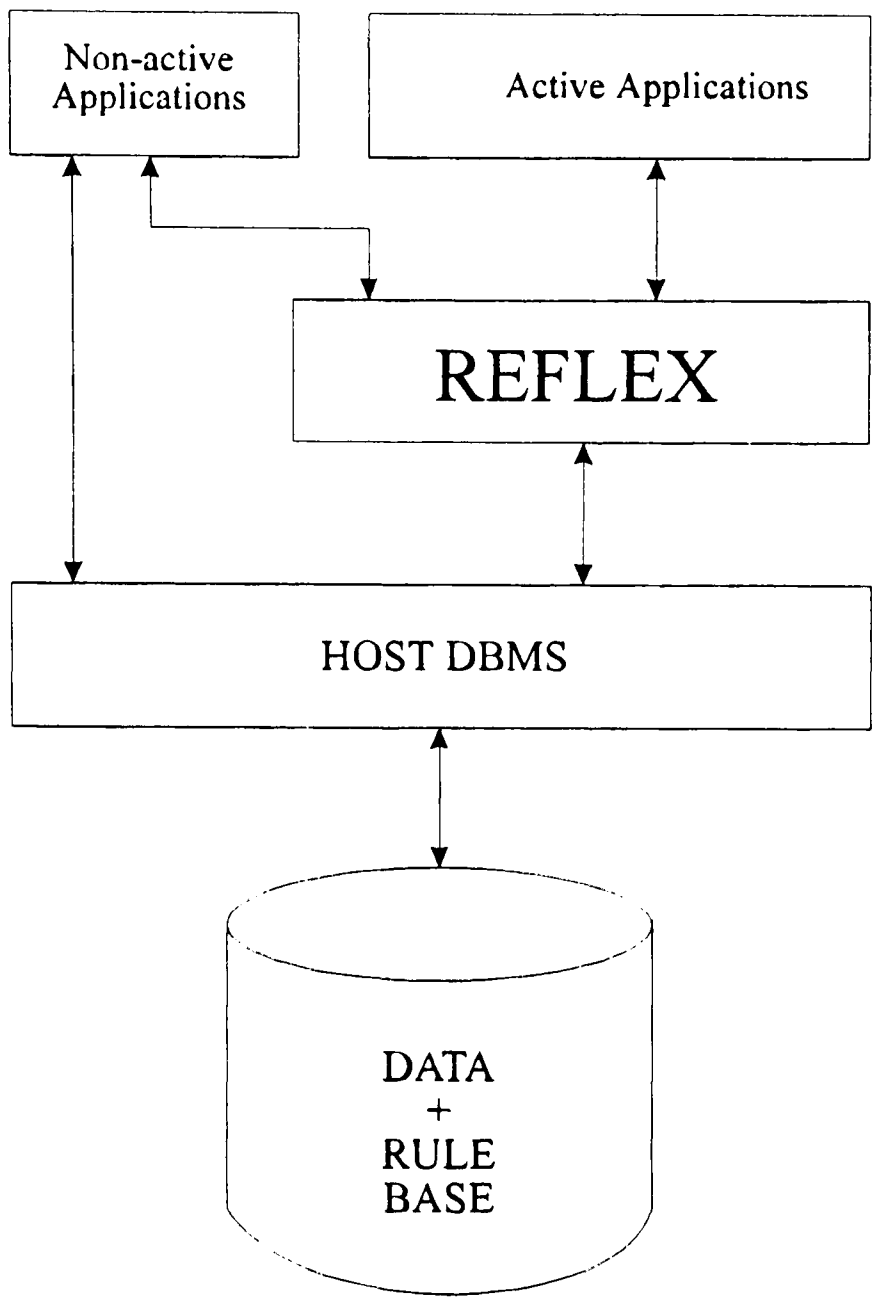


Figure 4.3 REFLEX active database approach

by the in-working-memory DBMS. Within the REFLEX model where knowledge coresides with data in the same database, even though the active knowledge extension is bolted-on to a host database, its knowledge facilities are tightly coupled, figure 4.3. This seems to be a paradox since the REFLEX extension is loosely-coupled w.r.t. host DBMS but tightly-coupled w.r.t. knowledge management, this affords a powerful solution to the problem of knowledge maintenance within a database and at reduced

overall system cost and satisfies the aim of portability.

It is this feature that should allow an organisation to utilise the advanced concept of active databases whilst still preserving its investments in technology and resources (training etc.), by continuing to use its existing database management system.

The engineering benefit of having a layered approach dictates that it may be implemented for any database and not just the one it was developed for. This satisfies its portability criteria enabling the system to be compiled for a different platform i.e. hardware, operating system and DBMS. Another more important goal is that of adaptability. This is where REFLEX adapts to its host DBMS in a transparent manner, allowing the system and its applications to function as before. This feature is investigated in depth later in chapter 6 (Design Architecture and Implementation).

4.4. Knowledge Model

Both events and rules are modeled as first-class objects within REFLEX, as is the case with ADAM [Diaz 91b]. Except that ADAM only allows an event to affect one rule and a rule can only be triggered by one single primitive event. REFLEX like systems such as HiPAC [Chakravarthy 89] and Ode [Gehani 92a], provides support for complex or composite heterogeneous events in addition to primitive events, allowing the user the flexibility of defining either a simple or composite event for a given rule.

REFLEX promotes the Extended ECA (EECA) knowledge model, which is an extension of the ECA [McCarthy 89] model. The EECA knowledge model addresses the problems associated with scope of the condition clause and situation redundancy. The constituent parts of the knowledge model are discussed in depth in the following chapter.

4.5. Execution Model

Like HiPAC, the coupling anchors afforded by REFLEX between the host transaction and the interrupting transaction are immediate, deferred or decoupled for the evaluation of the condition. REFLEX promotes an extended knowledge model for which the same modes are available for the execution of the multiple EECA action clauses.

4.5.1. Rule Contention

In order to comply with the portability design goal, the rule contention scheme for the knowledge management extension should be consistent on as many platforms as possible. In order for the rule contention strategy to be available on all platforms implies that the lowest common denominator be extracted from all possible platforms and implemented in REFLEX. Some platforms may be single-tasking, multi-tasking, uni-processor, or multiprocessor. The lowest common denominator would necessarily mean single-tasking/single-processor. Contention strategies for these systems (single-tasking) have generally meant rule priority mechanisms i.e. where the rule whose condition is satisfied first is allowed to execute and if two or more rules are satisfied then the rule with the highest priority will execute. This approach is satisfactory but may handicap the system when operating in an environment which supports multi-tasking, since it cannot take advantage of more than one processor. For this reason, REFLEX has a tiered or stepped approach. Where the user is presented with an interface which supports the multi-processing system, i.e. the user may set a priority level for the rule, but may also set a high 'trap' priority which instructs the system to execute in parallel.

4.5.2. Rule Termination

A problem for active databases is that of cyclic firing of rules where on the firing of a rule, a further event is raised which may indirectly cause the initial rule to be again fired. REFLEX attempts to prevent this situation from occurring by two methods (i.) prevention and (ii.) detection. The first preventive method attempts to minimize the correspondence of rules to only a few related rules. These rules are grouped into cohesive rule sets which reduce the scope of the rules to one scenario. Hence the rules should be more easily analyzed and the interrelationships minimized. The second more crude method is that of dynamic detection where on the firing of a rule, its firing count is stored against a situation. Once the maximum number of allowable firings have occurred for a given situation, the rule can no longer fire for that situation. The maximum number can be set by the user, but the system provides a default of 30.

4.6. Employing Activity

An active database provides a fast reaction to any changes within the database's state or the applications environment i.e. *imparting* active capability into the application domain. REFLEX, unlike any of the other active database research prototypes, *employs* the active capability itself i.e. it is *self-active*. The knowledge base (KB) as well as the application database are stored within the REFLEX system. Thus the maintenance of the KB can also be subject to the notion of activity. As an example, the rule's state is monitored *actively* by the REFLEX system. Rules have three components: events, conditions and actions. The clauses for each of these components are compiled, translated or recompiled at the point of rule creation or on rule modification. The re-compilation process being automatically triggered on a rule change.

The goal of REFLEX was to provide activity to a host database. Since a motivation was to allow the application domain knowledge to be centralized within the database,

and hence reduce maintenance overheads, why not allow REFLEX itself to utilise the activity to maintain itself. This *self-activity* feature was actively pursued in designing the system.

4.7. Knowledge Integrity

It may be a good goal striving to promote more knowledge within a database, but this knowledge should be audited to ensure that the system is reliable in its knowledge inferencing. REFLEX provides many features for the specification and testing of the knowledge entered.

As with expert systems which make inferences, the user needs to know that the inferences made are correct, given the known information. This is generally achieved by having an explanatory interface, which explains the rationale for the firing of certain rules.

Most expert systems are main memory based and have a finite number of rules (exceptions are systems such as XCON [Luger 89], built on OPS5 [McDermott 81], which according to Soloway et al. has a large and increasingly unmanageable set of rules [Soloway 87]). Active databases have knowledge, generally represented as production rules, but are based on large databases. This knowledge must exist for a long time, possibly indefinitely. A user may wish to know why a particular action took place last year, what were the conditions etc.? This then leads to the difference between manual and computer based systems. In a manual system, if a customer notified a company of a change of address, the piece of paper holding the new address is generally placed in the customers file or folder. The following year if the customer again moves, a piece of paper is again deposited in the file. The same scenario using a typical computer based system would mean that on receiving the customer's new address, it is entered over the customer's old address, destroying the previous

information by updating the record. A destructive update. Some systems, however, can be designed to handle more than one address, but how many? A lot of work is currently being undertaken into this field of temporal databases, for example the General Temporal Model by Knight [Knight 92a] and Ling and Bell's Temporal Model [Ling 92], where the data is not destroyed on every update. Akin to the old fashioned manual system. This approach is followed in REFLEX in order to maintain the knowledge base.

4.7.1. Non-Destructive Knowledge

REFLEX introduces the concept of *Non-Destructive Knowledge* i.e. declared knowledge is not lost. For example, if a rule has been declared, and it has not been used, it may be subject to change or amendment. But if the rule has been fired, or linked, it may no-longer be subject to change. It is in effect, locked. This concept allows us to audit our knowledgebase and evaluate why certain events occurred. It also allows the provision of *knowledge versioning*. If a change in the rule's definition is required, a new rule must be declared, which the old rule references. The rules, even if deactivated, still maintain references to objects that they referred to, thus providing a browsing system of the previous database knowledge state.

4.8. Summary

The research described in this chapter will provide an adaptive active data model for an existing database system. Therefore if an organisation has invested in technology and the training of its staff, the product of this research will allow the organisation to keep both. The existing databases may still be used, but the knowledge dimension, may simply be *bolted-on* as a certain application requires. Providing a very flexible cost-effective solution.

The following chapter reviews the REFLEX knowledge model. The adaptability and flexibility of REFLEX is reviewed in chapter six.

Chapter 5

The REFLEX Knowledge Model

Active databases have the ability to manage knowledge. This knowledge must be structured or modeled so as the semantics of rule operation are known. An active database is essentially an event-driven knowledgebase system. The events and their detection are therefore of central concern. This chapter describes the Extended ECA (EECA) knowledge model promoted by this research, including its handling of the problems associated with situation redundancy, the rule and event representation methods employed, and the event specification language known as **English ESL.**

5.1. Introduction

Before the Knowledge Model employed by REFLEX can be discussed, it would be best to define exactly what a knowledge model is. For the purposes of this research a knowledge model defines how the inherent knowledge within a system can be structured, represented, managed and utilised.

REFLEX's knowledge model determines the way the knowledge is defined and maintained. The knowledge model also defines the method by which events are modeled and handled. The execution model, which is a part of REFLEX's knowledge model,

implements the various available transaction coupling modes between the condition and action clauses of the rules.

This chapter is structured as follows: Section 5.2 presents an overview of the knowledge model and its components. This is followed by the new Extended ECA knowledge model which this research promotes. Within REFLEX, the rule is the primary method of knowledge representation employed. Section 5.4, describes how the rules are modeled as first-class objects. Events and their representation within the system are described in section 5.5. These are followed by the event specification and their semantics. Sections 5.8 and 5.9 declare the detectable events and how complex events are constructed by means of using the English ESL algebra.

5.2. Knowledge Model

REFLEX's Knowledge Model is based on similar lines to the ECA model [McCarthy 89], although it has been extended into the EECA model [Naqvi 94d] which will be discussed in the following section. The knowledge is represented as production rules [Williams 87] or simply *rules*. The production rule is a single *condition-action* pair and defines a single chunk of problem solving knowledge. The rule is brought into context on the occurrence of an event(s). At this point the *condition part* of the rule, which is a pattern that determines when the rule may be applied to a problem instance, is evaluated. If the condition is satisfied then the *action part*, which is the definition of the problem solving step, is executed.

The knowledge model can be defined as follows, figure 5.1. Rules apply to objects and an object may have many rules which apply to it. Rules can be assigned to classes or to individual instances of objects.

The rules may have one or more events defined, that may trigger them. This implies

that if more than one event can be defined against a rule, then the system (REFLEX) allows both primitive and complex event specifications.

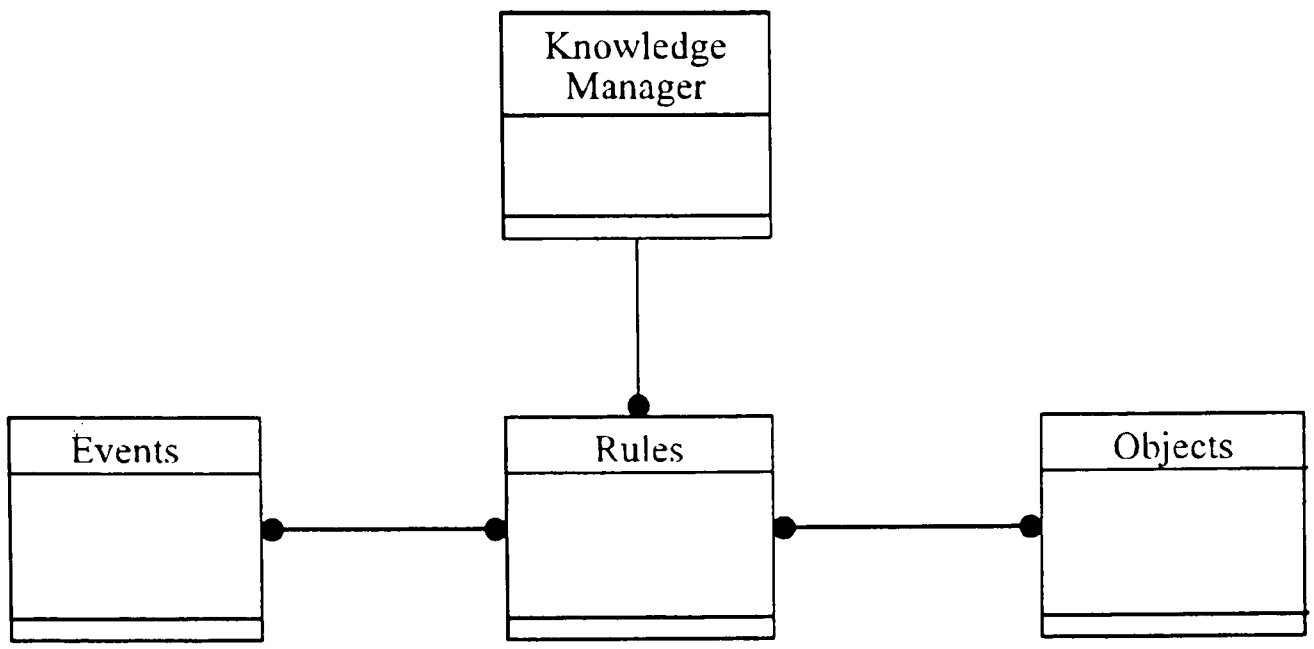


Figure 5.1 REFLEX Logical Knowledge Model

According to Dayal [Dayal 89], within any system there is almost certainly a point of control and this requirement becomes even more important with active or event driven systems. The REFLEX active database system has a kernel or control module, known as the Knowledge Management Kernel, to oversee the system and manage the scheduling tasks that are inherent in an asynchronous system. Within REFLEX, any application domain may have one and only one kernel, which is also modelled as an object.

The rules themselves belong to sets [Naqvi 93d]. A rule set is a mechanism used to group related rules together, primarily used to allow the analysis and auditing of rules and their interactions.

5.3. The Extended Knowledge Model

REFLEX was initially designed around the ECA model, and was proven using prototypes. These are described in chapter six. Applications (which are described in the appendices), were built to test the prototypes. These investigations highlighted several omissions of the standard ECA model, such as the replication of rules, and the creation of negative rules. These findings led to the Extended ECA (EECA) model which REFLEX now supports. This section discusses this EECA Knowledge Model.

5.3.1. Related Knowledge Models

A survey of active database systems and their knowledge models appeared in chapter three. In this section for the convenience of the reader, a précis is provided of some of the important knowledge models.

Most of the active database research prototypes use the Event-Condition-Action (ECA) model developed within the HiPAC project. This ECA model is now a dominant knowledge model used within the active database community e.g. it is used by StarBurst, POSTGRES, ADAM, etc.

Gehani, Jagadish and Shmueli propose an Event-Action (EA) model [Gehani 92a] for the Ode object database system, which combines both the event and condition clauses of the ECA model into the event specification. The rationale for this approach was based on the fact that Ode is an extension to C++, an object-oriented programming language. The aims of the extension are to provide persistence to C++ objects and database facilities such as transactions, recovery and security measures. As such, it is constrained by normal programming language development goals, many of which are *at odds* with those of database development i.e. database environments provide data independence and longevity of data, whereas programming languages provide

optimised static object code and take a short-term in-memory view of data. Gehani et al. [Gehani 92a] report that the ECA model provides too many coupling-modes which are difficult to maintain within a programming environment, and further state "*the E-A model is easier to explain and has simpler semantics than the E-C-A model*". Although the EA model does away with many of the coupling modes, as the event and condition clauses are now one, the current research has found the approach restrictive because in order to satisfy an event specification both the event and condition masks need to be evaluated, as discussed in chapter three.

The REFLEX EECA model addresses these problems, of situation redundancy (identical declarations of both the event and condition clauses), and the scope of the condition clause.

5.3.2. Scope of the Condition Clause

Most of the current active database prototypes allow the condition clause to be declared using some sort of Data Manipulation Language (DML) query. This form of condition declaration is recognised as useful, as it allows the user or designer to use a familiar interaction protocol. However, it is also limiting as it forces the designer to initiate unnecessary access to the database, thus adversely affecting the performance of the overall system. For example, for a large office complex management system, if the fire alarm sounded how would the active database know if it was a test or a real fire emergency, since the fire station should only be called on a real fire.

```
ON      Event Alarm
IF      select room
        from rooms
        where status = fire;
THEN    call fire department
```

The room information is probably held in the Alarm Control Box somewhere in the building. But how did the database acquire the room information in order to test its state? Since other active databases test internal conditions only, for the room information to be tested, an update to the database must be applied so that the data is in the database, i.e. the above form of the condition clause addresses *only one* aspect of the total environment, that is the *internal* state of the database.

REFLEX however, with its EECA knowledge model, allows the *calling* of user defined condition modules. This provides support for changes in the environment which may require a complex condition statement which cannot be handled by the DML language, or the condition requires access to external or application specific parameters, possibly user initiation, which have no bearing onto the internal state of the database. For the above example, the following rule could be declared:

ON	Event Alarm
IF	*call* getAlarmStatus
THEN	*call* fire department

The database calls the external getAlarmStatus routine, and thus avoids internal database updates to test the environment. The external condition module is recognised as it is preceded by the *call* keyword. This approach was taken to distinguish external conditions from internal object SQL statements signified by the *SELECT* keyword and to distinguish from the conditions specified in the proprietary language of the host DBMS, which are entered as normal without any specific pre-keyword. The external condition module simply returns a boolean TRUE if the condition statement is satisfied or FALSE otherwise.

This extension allows all the sections of the EECA tuple to independently access either internal or external factors of the environment i.e. the external events, conditions and actions.

5.3.3. Situation Redundancy

There may be situations (both events and conditions) which are common to many rules, but each with alternate actions i.e. the same situation in the environment triggers these rules. An example scenario could be in an office environment where there is a stipulation that working temperatures are to be within a defined range. If the room temperature is greater than the maximum working temperature a number of activities take place, (i.) for system security the system should be backed up, (ii.) the maintenance department must be informed and (iii.) the room should be evacuated. These three actions, under the ECA model require these rules as follows:

```
i.    ON      UPDATE room_details
      IF      temperature > maxTemperature
              AND airConditioning = "ON"
      THEN    run backup

ii.   ON      UPDATE room_details
      IF      temperature > maxTemperature
              AND airConditioning = "ON"
      THEN    call maintenanceDepartment (Room No)

iii.  ON      UPDATE room_details
      IF      temperature > maxTemperature
              AND airConditioning = "ON"
      THEN    call initiateEvacuateRoom (Room No)
```

If events are raised which bring into context many rules, the event specification clauses of these rules must be evaluated. After the event specification clause has been evaluated and satisfied, the condition clause must also be evaluated. If the situation of the rules, are the same, then it is implied that there has been multiple or redundant evaluation of

event and condition clauses from many rules, causing the overall system to be inefficient.

The proposed EECA model alleviates the problems associated with redundant situation declaration by allowing a rule to have multiple actions, each within their own transaction. Thus on the occurrence of a given situation, the rules' many possible actions may be executed. The multiple action clauses also implies that a rule must have multiple Condition-Action coupling modes. For the above example, an EECA rule could be declared as:

```
ON      UPDATE room_details
IF      temperature > maxTemperature
        AND airConditioning = "ON"
THEN    run backup
        call maintenanceDepartment (Room No)
        call initiateEvacuateRoom (Room No)
```

There are occasions where it is easier to state a *negative* condition rather than a normal condition, as it may be far more efficient to evaluate. The EECA model accommodates this situation by using a construct that is similar to an *else* statement in conventional block structured programming languages. For this case the EECA model proposes *Fail Actions*. These are actions that may be executed if the condition clause of the rule fails (or does not hold). Multiple fail action clauses as well as multiple action clauses are also permitted within the EECA model, along with their respective Condition-Fail-Action coupling modes.

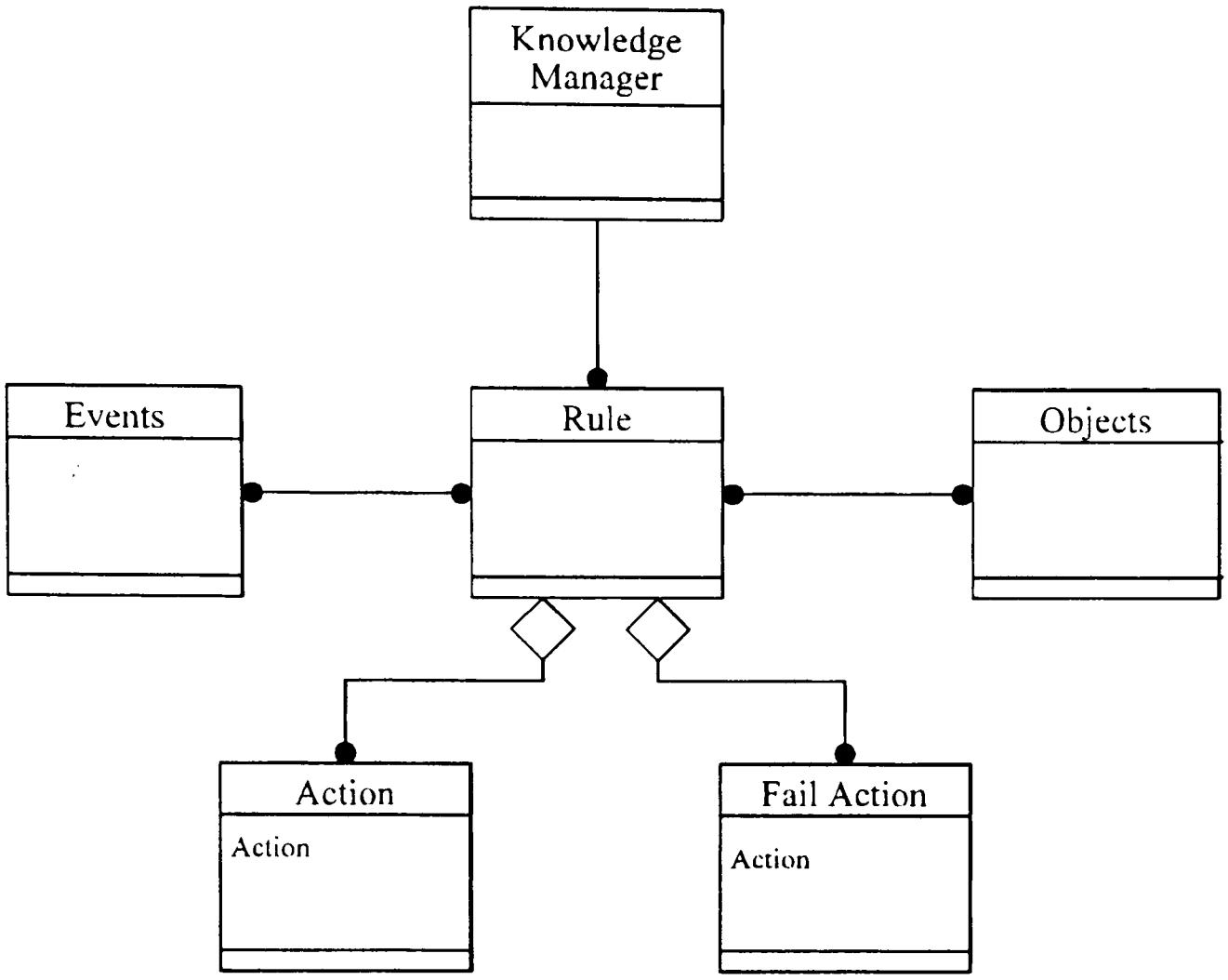


Figure 5.2 **EECA Knowledge Model**

A rule in the REFLEX Knowledge Model, figure 5.2, is represented as:

ON	event specification	
IF	i) internal:	NULL condition holds OSQL host DBMS prop. language
	ii) external	
THEN	multiple action clauses execute action 1 ... execute action n	
ELSE	multiple fail-action clauses execute fail action 1 ... execute fail action n	

The Action and Fail-Action clauses are mutually exclusive, just as with the THEN-ELSE structure. The clauses may contain requests to abort the parent transaction, undertake some DML query or call some external module.

5.3.4. EECA Coupling Modes and their Semantics

As described earlier, one part of the ECA triple defines how and when the subsequent part is actioned. This is termed the coupling between the two parts. In order to evaluate the condition, the event-condition coupling mode defines whether the condition clause is to be evaluated immediately, or deferred until the end of the host transaction or whether it should be evaluated within a separately spawned decoupled transaction.

With REFLEX's EECA model, there may be multiple action and/or fail-action clauses. In order for some autonomy to be maintained within the action clauses, each clause will require its own condition-action coupling mode. To these coupling modes, the complex issues of dependence need to be addressed, i.e. is the committal of the parent transaction dependent on that of its child?

Since flexibility was seen as an important goal for the REFLEX system, the onus for dependence between the parent and sub-transactions has been passed to the designer of the application system. The EECA model requires that all the action statements (including fail actions) for each of the rules have a dependency flag that signifies whether the action is dependent or independent of its initiating transaction. Hence the action clause is effectively an object or tuple (with arity 3), as is demonstrated below:

Action clause	(execute action 1, coupling mode, dependency flag)
	.
	.
	.
	(execute action n, coupling mode, dependency flag)

The same is true for the fail-action clause.

Fail Action clause (execute fail action 1, coupling mode, dependency flag)

.

.

.

(execute fail action n, coupling mode, dependency flag)

It may be noted that the EC coupling modes for the condition clause remain unchanged from those for the ECA model i.e. the condition clause can have one of the following coupling modes: immediate, deferred or decoupled.

For a given situation, where there may be many actions, an EECA rule could be declared using multiple action clauses but only if the EC coupling modes for the situation are also the same. If the EC coupling modes are different, then different rules need be declared.

e.g.

R1	ON	UPDATE student
	IF	student.name = "Joe"
	THEN	....
	EC Coupling Mode	immediate

R2	ON	UPDATE student
	IF	student.name = "Joe"
	THEN	....
	EC Coupling Mode	deferred

In the example above, two separate rules need to be declared since the EC coupling mode for the same situation is different. This design decision was taken so that the rule declaration was not over complicated with many excess coupling modes for situations which would hardly ever arise.

For the Condition-Action coupling the three modes (of immediate, deferred and decoupled), are offered the option to be dependent or independent of the parent transaction.

5.4. Rules as *First-Class* Objects

In some systems such as Starburst [Lohman 91] rules are modeled as extensions to SQL and are stored within the system catalogs. This approach aids an organisation to migrate to an active database system, since SQL is extended with rule declaration facilities and hence allows a lower learning curve. However, it does not allow extra information about the rule to be maintained.

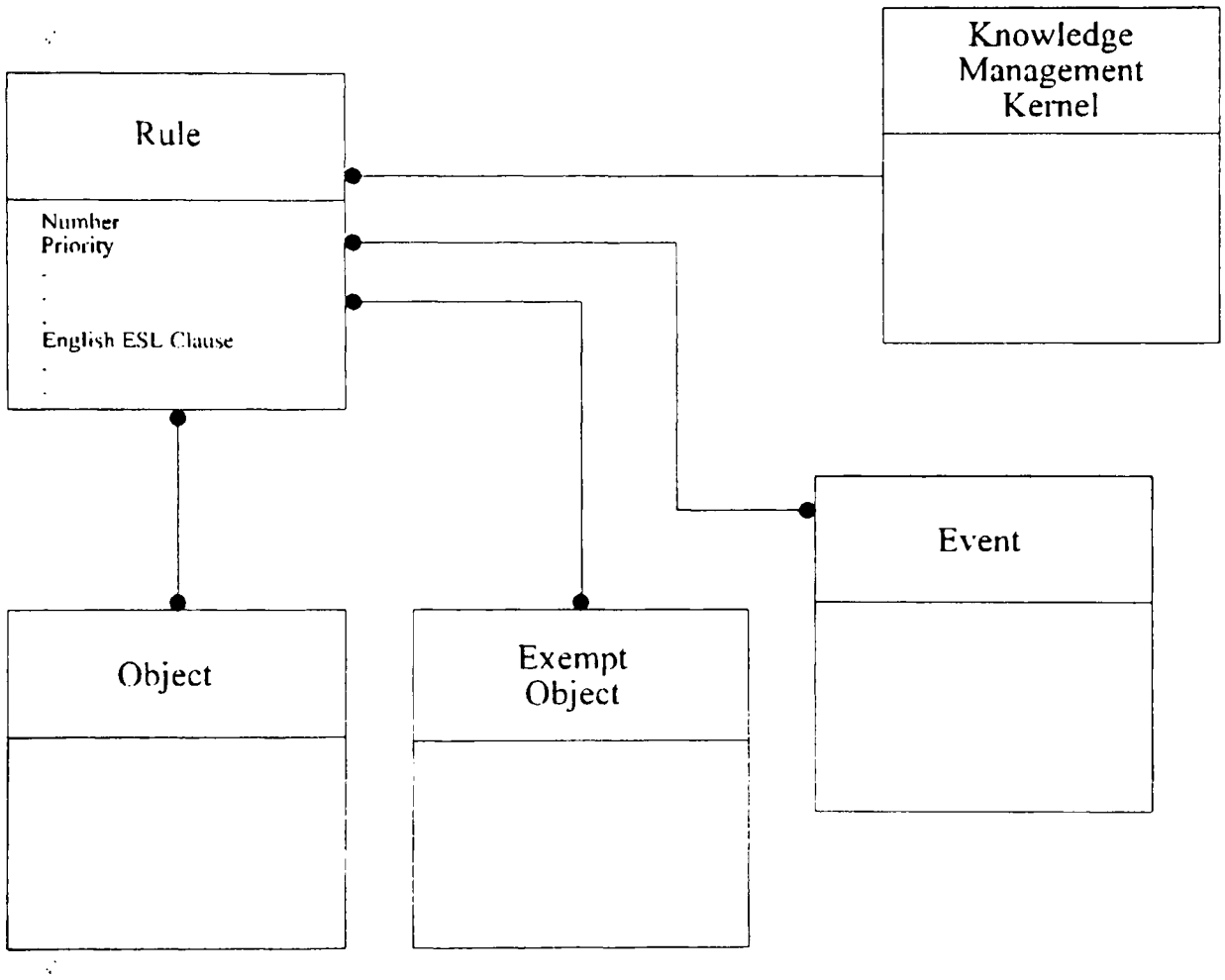


Figure 5.3 Partial Rule Composition Hierarchy

In the knowledge model embodied in REFLEX, rules are modelled as *first-class* objects

(objects in their own right), as in HiPAC [Chakravarthy 89] and ADAM [Diaz 91a, Diaz 91b]. This approach allows the rules to be handled in the same uniform manner as the other objects in the database and it has a number of advantages, the most important being that maintenance of the knowledgebase is simpler as the underlying DBMS maintains the rules as well as the data. Another important advantage is that the rules, which are objects of a Rule class, can be created during run-time at will. As soon as they are created they are immediately available for processing. If the rules were hard-coded into the application programs, they would have to be declared prior to compile-time.

The illustration in figure 5.3 shows, within REFLEX, a Rule as a first-class object. Some of its attributes can be seen but more importantly so can a portion of the complex object composition hierarchy. It is precisely this ability of aggregating objects into more complex objects which affords the object model more representation power over other systems such as the Relational Data Model. This allows the rule to be represented in a more natural and real-world manner since the rule encompasses not just the event, condition and action clauses but further attributes such as coupling modes and collections. These collections aid REFLEX by allowing the rules system to be efficient since a rule maintains links to the other objects which it is interested in, such as the central control object, the Knowledge Management Kernel (KMK). This link is a simple one since each rule is attached to exactly one and only one KMK. Links to the other objects can be multiple for example, a rule maintains a list of the events which affect it, and of the objects it rules upon.

5.4.1. Rule Attributes

The structure of the rules in REFLEX have the following main attributes, summarised in table 5.1:

- Knowledge Management Kernel (KMK)

  Each active application system must have only one central control point, the KMK. Each rule in a given application has a link to the KMK.

- Objects

  The rule maintains a list of all classes that it applies against, and to individual objects.

- Exemptions

  The object instances can also hold *exemptions* from certain rules as required. For example, in the case of a Student Records System, there will be a rule stating that students may register onto a course. In the case of a student who has been suspended, he/she will be exempt from the rule which allows registration. The registration rule will be at a class level i.e. on all students, and the exemption in this case, will be at an instance level, on a particular object.

- Event Algebra Specification

  The rule maintains the logical complex event in terms of an English ESL declaration. This specifies how the various component events are related together to form the logical complex event, using the event specification language introduced by this research.

- Events

  A list of the events that are specified in the English ESL clause, are maintained, primarily for efficiency and good house-keeping i.e. if an object refers to another object, then that object should maintain a reverse reference.

- EC coupling mode

There are different coupling modes between the event specification being satisfied and the condition clause being tested. The modes of immediate, deferred and decoupled are available.

- Condition clause

  The test of the state of the environment, either internal or external to the database is specified and stored in this attribute.

- Action clause

  A list of action clause objects is maintained, in a part-of relationship. The action clause objects have attributes to specify the action specification in a manner similar to the condition clause. The object also maintains the Condition-Action coupling modes of immediate, deferred and decoupled, and the dependency between the triggered transaction and the triggering transaction.

- Fail-Action clause

  As for the Action clause above. These clauses are triggered if the condition clause fails.

- Set Membership

  Each rule is a member of a set of related rules. This allows the interactions between rules to be monitored and minimized.

- Rule Priority

  Each rule has a defined priority. This is used during conflict resolution, where the rule with the highest priority is selected to action.

- isTrap

  This is a special flag which signifies that the rule has a special high

maximum priority status. Rules with this status are selected for concurrent evaluation of both their event and condition clauses.

- isActive

  A flag which may be set to indicate whether a rule is enabled or not.

- isTerminated

  A rule may no longer be available for being enabled. It is effectively dead, but its records are kept for auditing purposes.

- New Rule

  As part of the knowledge auditing, once a rule has been terminated, a link is maintained to the new succeeding rule.

- Old Rule

  As with New Rule, a link is kept to any previous incarnation of the current rule.

Rule	
Attribute	**Description**
Knowledge Management Kernel	Link to the nucleus of the system
Objects	List of objects a rule can act upon class and instances
Exemption	List of exemption instances of the rule
Events	List of applicable events
Event Algebra specification	English ESL - Complex Event Specification
EC coupling	Event-Condition Coupling mode
Condition clause	State Predicate Specification
Action clause	Link to multiple Action clauses
Fail-Action clause	Link to multiple Fail-Action clauses
Set Membership	Which Set the rule belongs to
Rule Priority	Priority
isTrap	Is the priority a trap
isActive	Rule Enabled or not
isTerminated	Rule is Terminated
New Rule	Link to new version of rule
Old Rule	Link to old version of rule

Table 5.1 Rule Object Attributes

The following sections describe the event representation employed within the REFLEX model.

5.5. Event Representation

As highlighted by authors such as Eswaran [Eswaran 76], Dittrich et al. [Dittrich 86],

events may trigger actions within a database. These events must be modeled and represented in an active database system. There are various ways of representing events within these systems. These are explained and investigated in this section, followed by the rationale for the chosen method of representation within REFLEX.

5.5.1. Events as Application System Attributes

HiPAC, according to Chakravarthy et al. [Chakravarthy 89] and Ode, as illustrated by Gehani, Jagadish and Shmueli [Gehani 92a], model events as application system attributes. The events are hard-wired into the system and their names are encoded into some name or attribute table, figure 5.4. This is the simplest and most conventional method of representing events within a system. It is however, a static method as events must be setup and declared within the source code at compile time. This provides fast execution and interpretation of events but, is an inflexible approach. What can a user or developer do once it is realized that a new event is required which does not exist in

```
// Database Internal Events        ***     0-60 RESERVED for system

#define REF_NullEvent               0       // Null Event Not used
#define REF_Write                   1       // Write to Database
#define REF_Update                  2       // Overwrite data
#define REF_Read                    3       // Read from Database
#define REF_Delete                  4       // Delete from Database
#define REF_LockWrite               5       // Lock item for write
#define REF_LockRead                6       // Lock item for read
#define REF_Lock                    7       // Lock item

#define REF_SysClosure              10      // SYSTEM Routine Closure

#define REF_TransStart              20      // TransStart
#define REF_TransStartAfter         21      // TransStart After
#define REF_TransCommit             22      // TransCommit
#define REF_TransCommitAfter        23      // TransCommit After
...
#define REF_ ...
```

Figure 5.4 Events as System Attributes

the system? New events may be added but the process is expensive, since, to add the new event, the underlying active database system code must be modified, and recompiled by an active database system programmer. These modifications are very costly in both monetary and system time dimensions. It may be infeasible to recompile a live database system since side effects may be unknown.

Another problem is that of operational efficiency, i.e. how long does it take to decide whether the occurrence of an event is of use to the system or not.

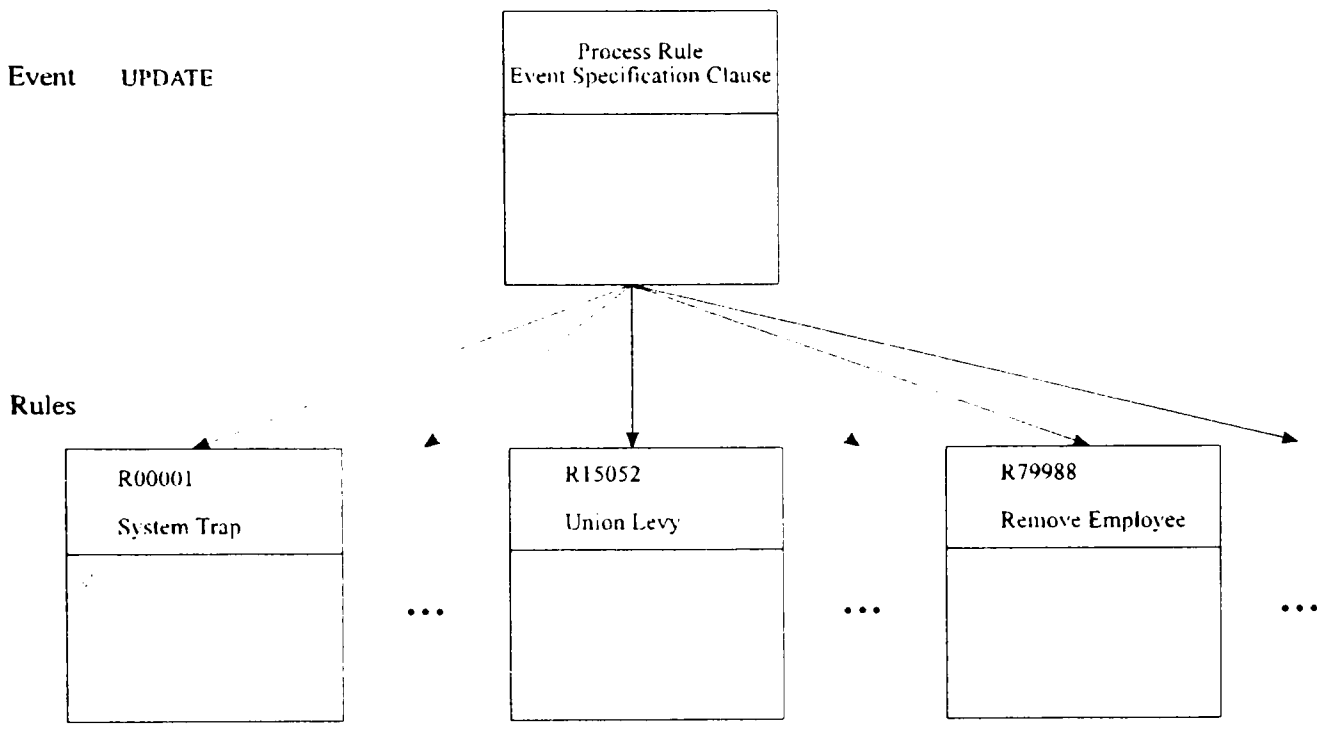


Figure 5.5 **Event as Attribute: all Rules in the system are processed**

Since the event is a flag in the application system, it does not normally hold any usage or reference knowledge (although this may also be represented). When an event is raised, the Knowledge Manager is given the event flag by the event detector. Since the event does not have reference information, figure 5.5, it must then process every rule in the system to establish whether the event affects it or not. This is a very expensive process. Operationally, indexes can be maintained, but these would be the responsibility of the DBMS and they would be external to the event.

To reduce the search space, the events can also be provided with knowledge of which

grouping of rules they may affect. This can be modeled by allowing each rule to be a member of a set. This approach implies that a rule can appear in any number of sets that an event affects, and that each individual event maintains a list of sets to which it may apply. To handle this scenario more powerful representation methods than application system attributes must be employed, such as, modelling events as first-class objects, the subject of the following section.

5.5.2. Events as *First-Class* Objects

Events may be modeled as *first-class* objects, as in ADAM [Diaz 91b]. This provides a uniform approach, as all components within the system are modeled in the same way, and hence the underlying system can maintain all of the components i.e. events may be created, deleted and modified as other data objects.

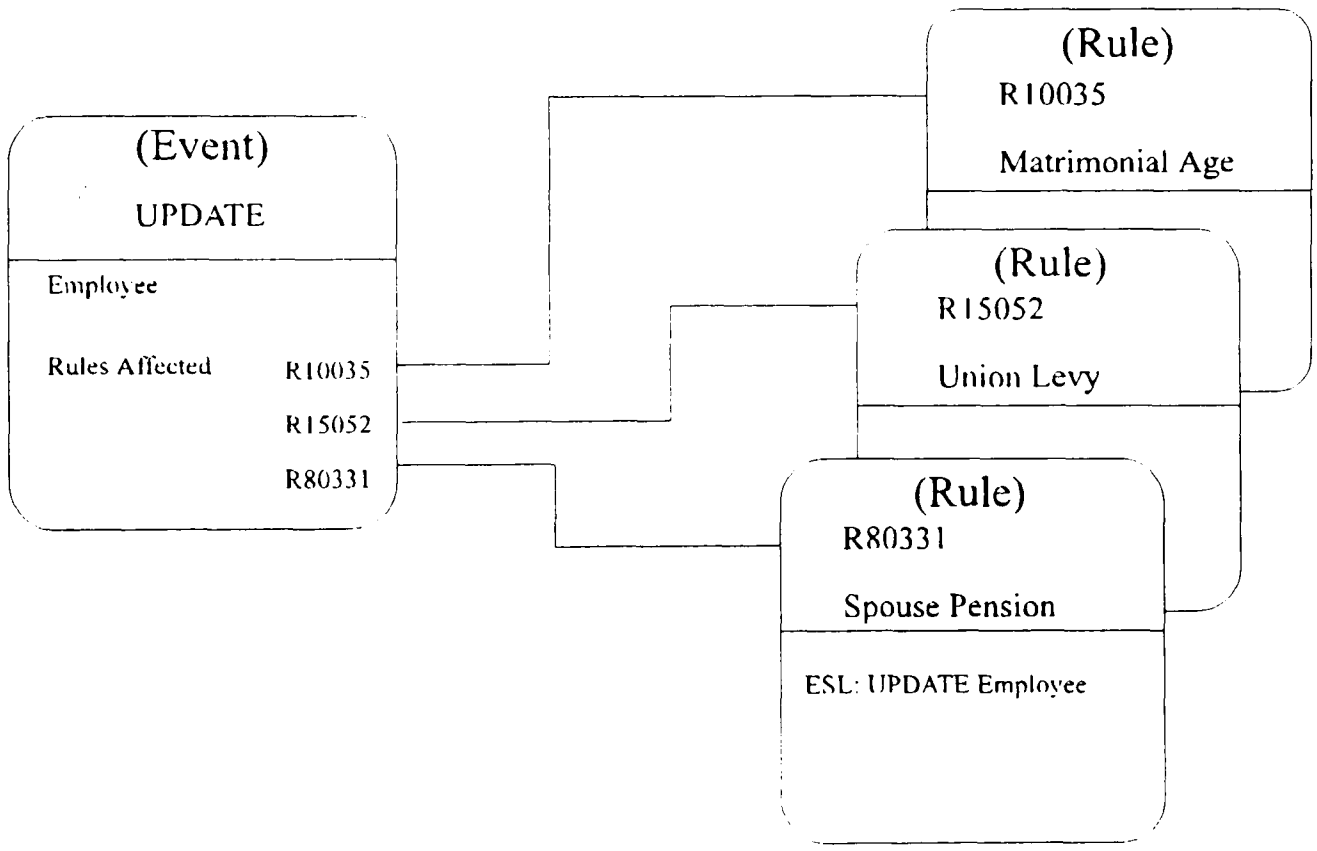


Figure 5.6 Event maintains list of rules which it may affect

Modelling events as first-class objects, on first analysis, may cause severe degradation

of service. This is because, on an event being raised, the event object is usually retrieved from the database, before its raise method can be called. Inherently, it seems to be plagued with intolerable overheads i.e. the time taken to seek the record in the host database, to retrieve it into working memory and finally to call its raise method.

This overhead can be countered by the utilization of the event object, which has access to standard object modelling techniques, the most important being the complex object facility. Each event can maintain a list of rules to which it may apply, figure 5.6. On the raising of any event, the Knowledge Selection Module (discussed in chapter six), has immediate access to the rules which are brought into context by the particular event. Hence, the system is much faster at sorting through its knowledgebase, on an occurrence of an event.

This feature becomes much more evident as the size of the knowledge base grows.

5.5.3. Complex events as first-class objects

Not only can primitive events be represented as first-class objects, but so can composite (complex) events, figure 5.7. This can lead to a scenario where the same composite event can be used as the event specification to more than one rule.

This approach does at first glance look rather elegant as an event is simply sub-typed into simple or complex, but it does cause several problems. Such as:

- The complex event must be evaluated, before any referenced rules are brought into context for their condition clause evaluation

- If the same complex event occurs in many rules, can the part-satisfied event specification clause be monitored for all the rules? The event

specification may be at different stages for different rules. These stages need to be tracked, which would be cumbersome and complex and lead to an increase in the overall overhead of the system, for very little gain.

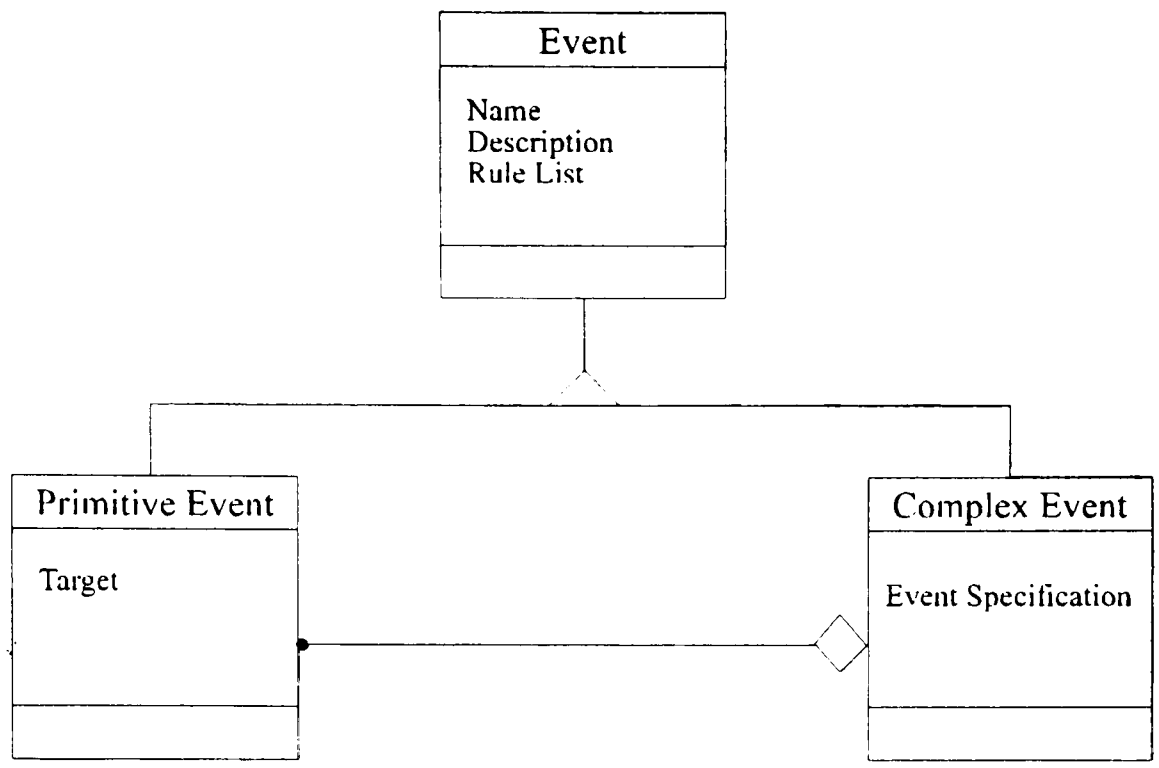


Figure 5.7 Events as complex objects

To model complex events as first-class objects introduces an extra level of indirection. Complex events can be seen as logical events made up of a number of primitive events, combined using an algebra. Whether the algebra declaration appears in the Complex Event object or in the Rule object is immaterial, albeit the complex event object conforms to a uniformity goal. The algebra still has to be parsed, the component primitive events satisfied. The Rule object is effectively the triggering complex event.

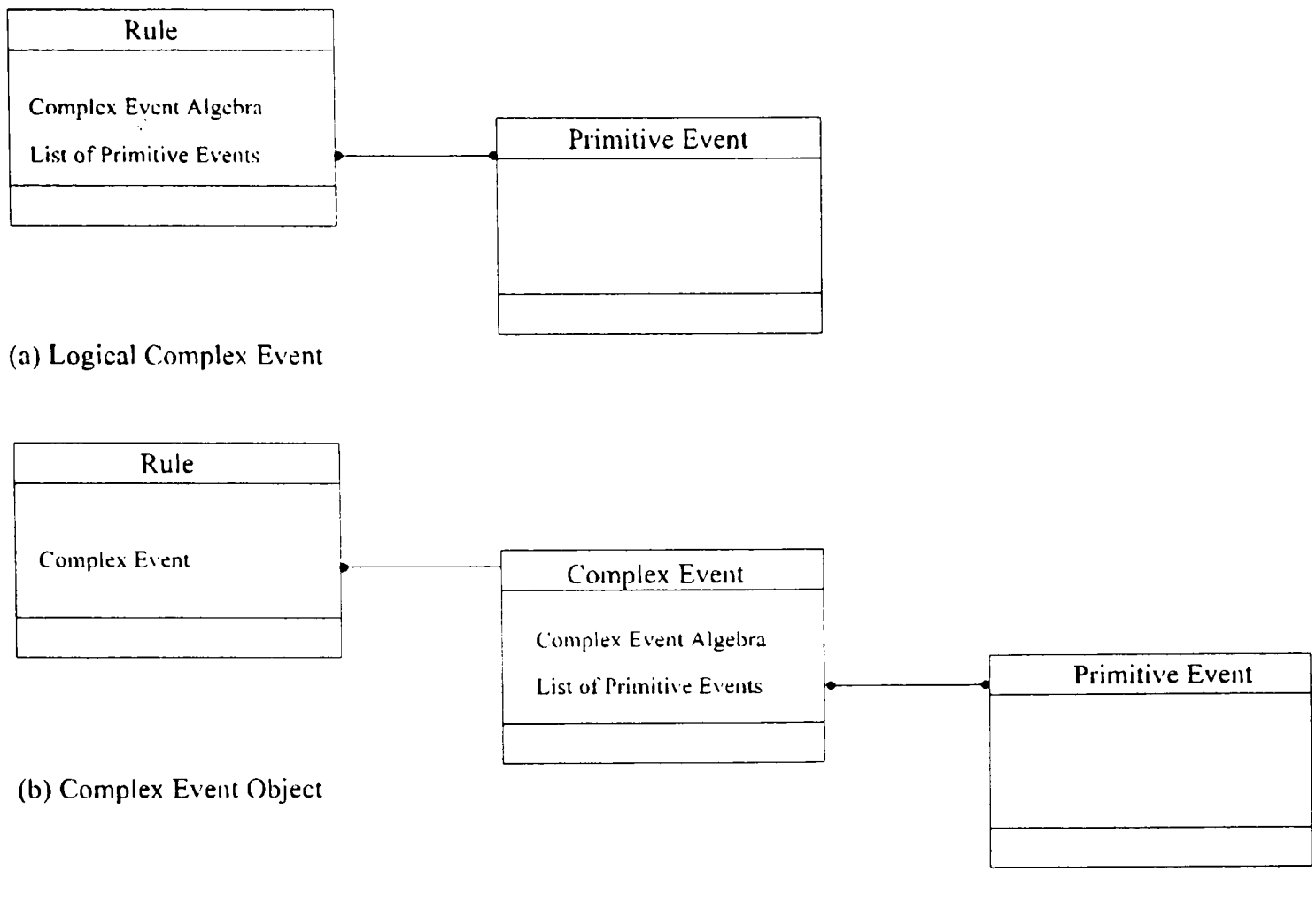


Figure 5.8 Complex Event levels of indirection

Figure 5.8 illustrates the extra level of indirection introduced by modelling complex events as first-class objects.

5.5.4. Event Representation Method Employed

REFLEX has adopted the method of modelling primitive events as first-class objects [Naqvi 92, 93a, 94d]. This decision was taken, as discussed earlier, because of design and operational concerns such as uniformity, maintainability and efficiency. If an event is represented as a first-class object, it can then be maintained in the same uniform manner as all other objects within a given database system.

This approach has allowed REFLEX not only to maintain the events in the system, but it also enables the developer to create events at will, at run-time. This feature is unique

to REFLEX and Adam [Diaz 91b]. But for Adam it could be argued that this ability of being able to declare events at run-time has been supported because of a side-effect of their development environment rather than actually being designed in, i.e. the environment is Prolog which is essentially interpreted at run-time, rather than C++ which is more mainstream and compiled.

The second goal of efficiency is served by the fact that REFLEX events can maintain lists of the actual rules that they may affect. This allows only the affected rules to be retrieved, without any wasteful searching. This is again unique to REFLEX. This is borne out by other systems such as Sentinel [Chakravarthy 93] and Samos [Gatziu 93], which model complex events as first-class objects, both of which report increased run-time overhead of modelling events as objects.

This may be illustrated by way of the following analysis.

5.5.4.1. Heuristic Analysis

To exemplify the concept that modelling events as first-class objects can improve system efficiency the following simple analysis is provided.

If a system has 1000 rules, it is likely that it may have approximately between 1 and 100 events of interest. Lets assume the system has 50 events. We can further assume that on average an event may affect up to 20 rules[1].

If events are to be modeled as system attributes, then on the occurrence of an event, all 1000 rules will have to be accessed to establish whether the event affects them or

---

[1]It is worth noting that from the panel discussion at the RIDE-ADS'94 workshop [Widom 94], the expert panellists stated that applications that were "*anything remotely complex* e.g. more than 7 rules", would not be supported by active databases in the near future.

not.

If on the other hand, events are modeled as first class objects, and maintain references to the rules that they affect, only 20 of the 1000 rules need to be accessed. This can be exemplified as:

$$\frac{number\ of\ rules}{tal\ number\ of\ rules} = percentage\ rules\ process$$

$$\frac{20}{1000} = 0.02$$

As can be seen from the above, only 2% of the rules needed processing, using the approach of modelling events as first-class object. Modelling events as system attributes, and using a centralized search for affected rules, causes an over processing of rules by 98% i.e.

$$1 - 0.02 = 0.98$$

It was decided not to represent complex events as first-class objects since the only real benefit would be the ability to declare a complex event which would bring many rules into context. This situation is handled by REFLEX's EECA model and its ability to support situation redundancy. The complex event is a logical concept represented as an event specification for a rule in REFLEX's English ESL, discussed in the following section.

5.6. Event Specification

The ability to respond to an event automatically is paramount in active database technology, for it is the event that activates or '*awakens the sleeping giant*' [Yamamoto 41]. It is one thing to respond to an event such as a clock tick, but totally different

when complex events are specified. These events occur over time and hence have a history. This section introduces the event specification language, known as **English ESL**, forwarded by this research. The language is compared to other languages proposed by related research.

What exactly is an event? An event is generally considered as something that occurs instantaneously, at a point in time. This definition is simplistic and provocative as there has been much research into the definition of time i.e. is time modeled as a set of points, as enunciated by McDermott [McDermott 82] in his temporal logic, or as intervals such as the theory put forward by Allen in his Interval Logic [Allen 81, 84], or a combination, such as the General Temporal Model of Knight and Ma [Knight 92a]. It is beyond the scope of this work to investigate the nature of time. Even so, time is an important consideration when the occurrence of events needs to be charted. Once again, an event can be considered a point in time since points in time for which some reaction is required, are of interest. These points must be specified in some way, such as the beginning or end of a database transaction, or explicitly, such as at 5pm. But what of the case where complex events require specification and detection.

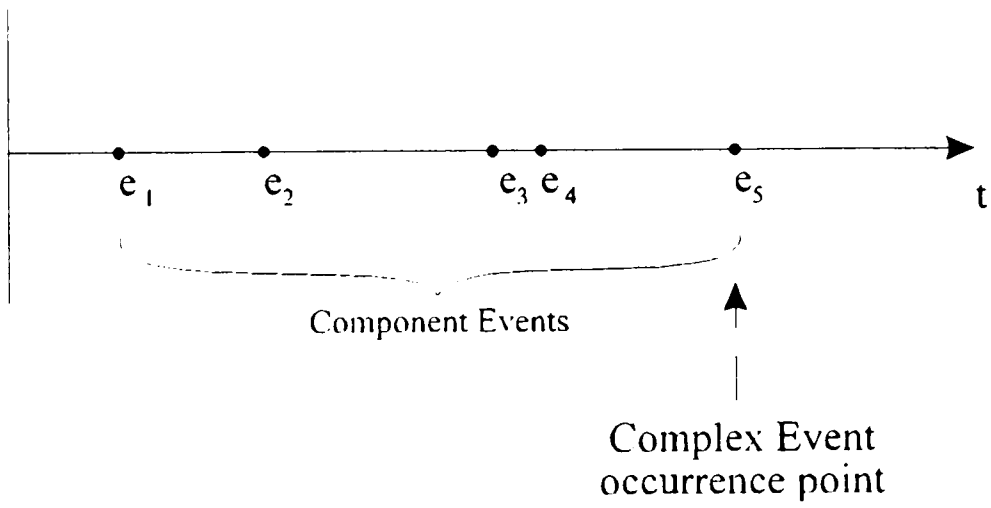


Figure 5.9 Complex event occurrence point in time

In this case, the point in time for the occurrence of the complex event is the point at which the last component event has occurred. This can be seen in figure 5.9 where the complex event e_1 *and* e_2 *and* e_3 *and* e_4 *and* e_5 can be seen to occur at the point of occurrence of the final requisite event, e_5.

The specification and representation of events are the subject of the following sections.

5.6.1. Related Work

There has been much work on event specification languages such as the logical model by Beeri and Milo [Beeri 91], Sentinel [Chakravarthy 91] and SAMOS [Gatziu 93], but the most widely cited work has been that embodied in Ode [Gehani 92a, 92b].

As enunciated in chapter three, instead of the typical E-C-A knowledge model, Ode has folded the event and condition clauses into one, resulting in an Event-Action (EA) model. This may seem natural as events are after all, a type of condition (they simply occur instantaneously as opposed to holding over time). But this approach at an implementation level can cause inefficiency, as described in chapter three.

The event specification language proposed by Ode, allows the declaration of complex events. Being based on the EA model, the declaration combines both the event clause and the condition mask in one. It allows internal (database and transaction), temporal (absolute, periodic) and logical events to be specified.

The event specification languages of the aforementioned systems, although there are some differences, are quite similar to those promoted by REFLEX but they differ in two important ways:

- Detection and verification of event specifications.
  The method used for the event detection is different, Sentinel uses an event graph, Ode uses a form of finite automaton and Samos a Petri Net. REFLEX uses an enhanced labelled Petri Net [Naqvi 93b] for its event detection and also for its system verification. These can be found in appendix C.

- Declaration language.

  REFLEX promotes a simple to use, easy to comprehend end-user language, English ESL, whereas the other systems are still declaring their complex event specifications using more mathematical and logic oriented declarations. For example, the WITHIN validation of an ordered conjunction (as described in section 5.8.2.5) is specified as follows:

  - REFLEX

    e_1 PRECEDES e_2 WITHIN t MINUTES
  - SAMOS [Gatziu 93]

    (E1 ; E2 IN [occ_point(E1)+01:00])
  - Ode [Gehani 92b]

    sequence (E1, E2) (WITHIN not supported)

A more general purpose approach is Kowalski's event calculus [Kowalski 86], which was developed as a theory for reasoning about events in a logic-programming framework and seems to be an appropriate foundation for a temporal event algebra [Kowalski 92]. It is based on the situation calculus of McCarthy [McCarthy 63, 69], but focuses on the concept of an event as highlighted in semantic network representation of case semantics. It does not however, seem to apply well to the domain of event occurrences in the form that are of interest to active databases, since it really looks at state changes as events. This can be explained because within an active database the specification and detection of an event is critical as it is the occurrence of the event, which then activates the database, and allows any testing of its state. The state of the database is a secondary concern. Hence Kowalski's event calculus does not seem appropriate as a foundation for an active event algebra.

5.6.2. Semantics of an Event

This section explains the concepts and operations of events within the REFLEX knowledge model.

5.6.2.1. Event Chronology

The *ON* or *event* specification clause of the rule allows both *primitive* (simple) or *complex* (compound) events to be specified. The complex event clause is expressed using an event algebra, which expresses the temporal relationship between the component events. Since complex events are composed of a number of primitive events, which each occur at different instances in time, these occurrences must be recorded. In effect the events have chronologies or histories which must be referred to in order to satisfy the event clause. This is the primary purpose of the *temporal log* [Naqvi 93b] to be discussed in chapter six, in which each occurrence of an event is recorded. Most active database systems that are capable of specifying complex events, such as HiPAC, Ode, Sentinel etc, provide support for some type of event chronology.

5.6.2.2. Internal Event Intervals

The temporal model employed within REFLEX is one in which an event is regarded as occurring at an instant in time. Emphasis is laid on the point of occurrence. This view is restrictive for some types of events i.e. internal. For some scenarios it may be important to specify a point of occurrence for a primitive event just before or after it actually takes place. For example, if a new customer wishes to purchase an item, the customer details would be captured, and just before committal of the details a new customer number would be assigned to the customer. If the number was assigned to soon, the customer may have changed his/her mind and decided against the purchase, and caused a customer number to be issued by mistake, which would then be lost. Hence, a facility is required to issue the customer number '*just-in-time*', i.e. just before

committing to the database.

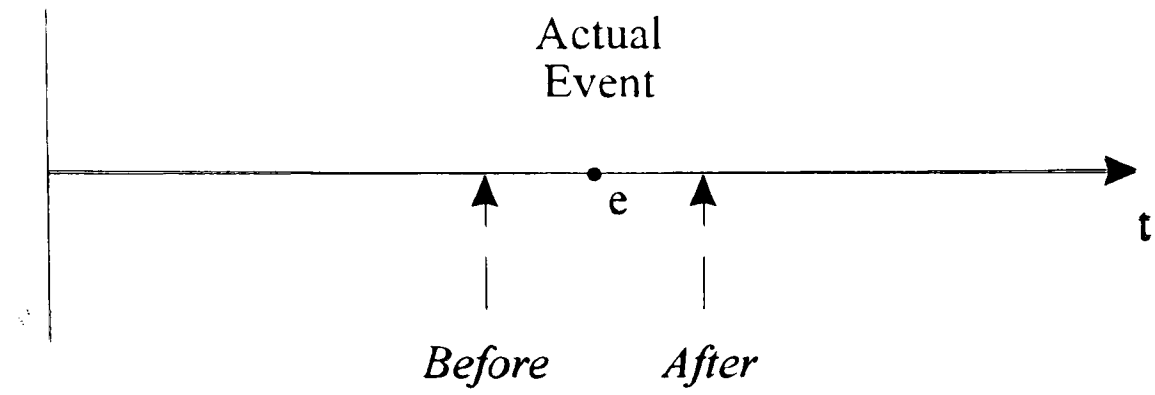


Figure 5.10 Event occurrence interval

Hence events being point based have a form of *interval logic*, with all internal events having a before/after granularity. All internal events generate a signal just before the event actually takes place and again just after it has taken place, as illustrated in figure 5.10. This means that the temporal system is discrete, i.e. there is a "next" point for every point.

The semantics of the event specification of internal events are that each event is preceded by either a *before* or *after* statement. If no mention is made, then *before* is assumed.

```
ON        before delete department
IF        select  e.Name
          from    employee e, department d
          where   d.DepartmentNo = Event Dept.
                  and e.DepartmentNo =
                          d.DepartmentNo;
THEN      Abort
```

Figure 5.11 Referential integrity check

This allows an application designer to trap various conditions, and preserve constraints. For example, if a DELETE Department operation was being undertaken, just before the actual delete was committed to the database, a referential integrity check could be performed to ensure that no employees were currently recorded as working for that

department.

A rule to enforce referential integrity, as in the above example, could be as in figure 5.11. This is obviously based on relational style set-at-a-time query, and has tested that the actual department that raised the event is tested for, which otherwise could be expensive if the rule were called on every delete department command. But, referential integrity checking is important, when you delete a department it should be clear that no employees are still working there.

Obviously, if the same rule were declared for an object-oriented system, the state test would simply query the department complex object to see if it had any employees attached to it, hence it would not be as expensive as for relational systems.

5.6.2.3. Validity

A raised event may not always be valid even though it appears in a rule's event specification. This can be explained by the following example.

> Lunch of 1 hour may only be taken between the hours of 12pm and 2pm. The following may be specified. Tom *may go* to lunch during the lunch period only *after* Harry *has returned*. If Harry *does not* return *within the lunch period*, then Tom *cannot go* to lunch.

In the example above, Tom may not go to lunch if Harry does not return within the specified time. Hence, REFLEX introduces the concept of *event validity* [Naqvi 92, 93d]. The event may have to occur within a specified time or in some particular sequence to be valid. An EECA rule for the above example could be as follows:

ON	staff.goneToLunch AND staff.returnFromLunch WITHIN 1 HOUR AND staff.returnFromLunch BETWEEN 12:00 - 13:30
THEN	goToLunch

ELSE noLunchYet

A similar concept of monitoring intervals has since been introduced in SAMOS [Gatziu 93], but its specification is more cryptic than that supported by REFLEX. For example, in REFLEX

e_1 SUCCEEDS e_2 WITHIN 24 HOURS

says that e_1 follows/succeeds e_2 within 24 hours. The same specification in SAMOS would read

(E2;E1 IN (occ_point (E1) + 1440:00])

In the case of a primitive event, if it is raised then it must necessarily bring any rules for which it is a simple event into context. For example if a rule had an event UPDATE PERSON then on update person the event is valid and the rule's condition clause can then be evaluated.

This is not the case for complex events since they are composed of more than one primitive event. They are related by some form of algebra (**English ESL** in the case of REFLEX). For example:

$Event_1$ AND $Event_2$ (WITHIN 30 MINUTES)

In the above example $Event_1$ and $Event_2$ must occur within 30 minutes of each other, regardless of sequence.

5.7. Detectable Events

Active databases react to some occurrences of interest. These occurrences of events have been highlighted in chapter three, and the events which are detectable by REFLEX are summarised below. They are grouped by the three main types of events

i.e. those internal to the database, temporal or clock-based and the externally generated events. They are given generic names i.e. the internal object event *get* could be a *read, retrieve, view*, etc. dependant on the underlying host DBMS.

Internal:

Object events:	before/after create
	before/after get
	before/after update
	before/after delete
Transaction events:	before/after start
	before/after commit
	before/after abort

Temporal events:

absolute (on a specific-date, at a specific-time)
relative (to an event occurrence)
periodic (repeat-after-period)
delay (wait duration)
sequential (time ordered conjunction)

External events: These events are application defined (or *abstract*) and hence cannot be listed. Examples could include the raising of a fire alarm or a pulse from a radar.

Once detectable events have been defined, their use i.e. activating rules, must be specified. If a complex event is required which is made up of a number of primitive events, it must be constructed using an event algebra. The following section introduces REFLEX's event algebra, the English ESL.

5.8. English ESL - An Event Algebra

The temporal event algebra used by REFLEX provides comprehensive constructs for specifying complex events. Unlike specification systems such as those proposed in HiPAC [Chakravarthy 89], Ode [Gehani 92a], Samos [Gatziu 93], Sentinel [Chakravarthy 93] ease of use has not been compromised as standard English statements are used to declare the powerful clauses.

The language has been designed so as to be as natural and English-like as possible, following COBOL's tenet but in terms of today's human computer interaction psychology. The keywords provided by English are in four categories: logical, temporal, internal and external.

The algebra contains several logical and temporal operators. The syntax and keywords are introduced in the following section, followed by their operational semantics.

5.8.1. ESL Syntax

- **Logical Operators**

unordered conjunction	E_1	**AND**	E_2
inclusive disjunction	E_1	**OR**	E_2
negation		**NOT**	E
time ordered conjunction	E_1	**PRECEDES**	E_2
	E_1	**SUCCEEDS**	E_2

- **Non-temporal Internal Operators**

  Before, just before the actual non-temporal event

  BEFORE E

  e.g. BEFORE UPDATE person

BEFORE COMMIT

After, just after the actual non-temporal event

AFTER E

e.g. AFTER CREATE person

AFTER ABORT

Note: AFTER DELETE class, is not supported

- **Temporal Operators**

Validity, a temporal limitation on a conjunction of two non-temporal events

WITHIN **number of HOURS/ MINUTES/ SECONDS**

e.g. E_1 AND E_2 WITHIN 3 SECONDS

(E_1 AND E_2) PRECEDES E_3 WITHIN 45 MINUTES

Periodic, a repetition of a temporal event from the current time

EVERY **number of HOURS/ MINUTES/ SECONDS**

e.g. EVERY 5 MINUTES

UPDATE document OR EVERY 10 MINUTES

Relative, a temporal event is raised after a specified delay from the current time

DELAY **number of HOURS/ MINUTES/ SECONDS**

e.g. DELAY 4 HOURS

Absolute, a temporal event is raised a specific point in time, or between a range

ON DATE **dd/mm/yy**

e.g. ON DATE 16/3/93

UPDATE student ON DATE 6/3/94

AT TIME **hh:mm**

e.g. AT TIME 5:00

ON DATE 16/5/95 AT TIME 13:30

BETWEEN **range date|time - date|time**

e.g. UPDATE student BETWEEN 16/3/95-28/3/95

General, temporal quantifiers

YEAR|MONTH|DAY|HOURS|MINUTES|SECONDS

- **External Events**

EVENT the event keyword precedes abstract or user-defined (external) events.

There is a precedence order of operations as with mathematics where multiplications/ divisions, followed by additions/subtractions. Each of the logical operators has a position in the order. The highest position being the NOT which is evaluated first, followed by PRECEDES, SUCCEEDS, AND and OR.

Parenthesis may be used to override operator precedence (using left associativity), or simply to improve the clarity of a long and very complex event specification.

Further examples of the English event specification language (ESL) are:

read student

simple internal event - read

before update account **or after** update employee

disjunction of two non-temporal events

Event$_1$ **precedes** Event$_2$ **within** 24 **hours**

validated ordered conjunction.

every friday **at time** 5:00pm

periodic

event radar

user-defined or abstract

5.8.2. Operational Semantics

For this research the approach taken by Knight [Knight 92b] in his discrete time system has been adopted as a formal foundation. The semantics of this may be defined as a discrete infinite set of points on a linear time domain. This maps well to the concept of events which occur at instances in time, and is illustrated on the following pages. The main properties of this formal temporal model may be summarised as follows:

- it consists of a well-ordered discrete set of elements T, which are points at which events can occur
- a total order may be defined on T and is denoted by $<$ and hence the events $e_1 < e_2$ may be interpreted as e_1 *occurs_before* e_2
- the immediate successor under this order relation is denoted by the *next* relation, and so *next*(e_1,e_2) denotes that e_2 is the immediate successor to e_1

  next (t_1, t_2) may be defined for $t_1, t_2 \in T$:

  next $(t_1, t_2) \Leftrightarrow t_1 < t_2 \wedge \nexists t.\ t_1 < t,\ t < t_2$
- the predicate *event (e, t)* is used to represent the connection between the actual event e, and the time of its occurrence t.
- a mapping D:T $\rightarrow$ R is defined. D(t) gives the time of the point t, the duration of time between t_0 and t where t_0 is some defined origin.

The formal system can be used to define operations within the event specification language. The syntax for its use is as follows:

(i) [non-temporal condition (e_1, e_2, ..., e_n)], e_{eval}=rule-evaluation(r)

(ii) event (e_1,t_1), event(e_2,t_2), ..., event(e_n, t_n); event(e_{eval}, t_{eval})

(iii) f(t_1, ..., t_n; t_{eval})

This specifies the time, t_{eval}, for the evaluation of rule(r). For example, taking the example in figure 5.11, the following could be declared:

(i) e_1 = delete department(d), e_{eval} = rule-evaluation(r)

(ii) event(e_1, t_1) event(e_{eval}, t_{eval})

(iii) next(t_1, t_{eval})

...

rule(r): select e.Name ....

According to this specification, rule(r) will be evaluated at time t_{eval}, where t_{eval} is the next cycle following time t_1 and where t is the cycle on which department(d) was deleted.

Using the above formalism, the semantics of the complex events formed using the ESL operators are as follows:

5.8.2.1. AND

An unordered conjunction of two events e_1 and e_2 is said to take place when both of the events e_1 and e_2 have occurred, irrespective of the sequence of occurrence, and time of occurrence. This may be stated as follows:

e_1 AND e_2

An example English ESL declaration for an unordered conjunction as defined above

could be:

UPDATE student AND COMMIT

where e_1 is the internal object event, update the student class at the point of its committal and e_2 is the internal transaction event, commit transaction.

Formally, e_1 AND e_2 is expressed as:

i. [non-temporal condition (e_1,e_2)], e_{eval}=rule evaluation(r)

ii. event(e_1, t_1), event(e_2, t_2), event(e_{eval}, t_{eval})

iii. [next(t_1, t_{eval}), $t_2 < t_1$] OR [next(t_2, t_{eval}), $t_1 < t_2$]

5.8.2.2. OR

Disjunction of two events e_1 and e_2 is said to take place when either one of the events e_1 or e_2 has occurred. This may be stated as follows:

e_1 OR e_2

An ESL example could be:

UPDATE student OR DELETE student

Formally, e_1 OR e_2 is expressed as:

i. [non-temporal condition (e_1,e_2)], e_{eval}=rule evaluation(r)

ii. event(e_1, t_1), event(e_2, t_2), event(e_{eval}, t_{eval})

iii. next(t_1, t_{eval}) OR next(t_2, t_{eval})

5.8.2.3. PRECEDES

An ordered conjunction of two events e_1 and e_2 where both of the events e_1 and e_2 have occurred, but e_1 occurs before e_2. This may be stated as follows:

e_1 PRECEDES e_2

An example English ESL declaration for an ordered conjunction as defined above could be:

CREATE student PRECEDES DELETE student

Formally, e_1 PRECEDES e_2 is expressed as:

i. [non-temporal condition (e_1,e_2)], e_{eval}=rule evaluation(r)

ii. event(e_1, t_1), event(e_2, t_2), event(e_{eval}, t_{eval})

iii. [next(t_2, t_{eval}), $t_1 < t_2$]

5.8.2.4. SUCCEEDS

For completeness the succeeds operator is also supported which is an ordered conjunction of two events e_1 and e_2, the opposite of precedes, where both of the events e_1 and e_2 have occurred, but e_1 is the successor to e_2. This may be stated as follows:

e_1 SUCCEEDS e_2

An example English ESL declaration for an ordered conjunction as defined above could be:

CREATE student SUCCEEDS CREATE person

Formally, e_1 SUCCEEDS e_2 is expressed as:

i. [non-temporal condition (e_1,e_2)], e_{eval}=rule evaluation(r)

ii. event(e_1, t_1), event(e_2, t_2), event(e_{eval}, t_{eval})

iii. [next(t_1, t_{eval}), $t_2 < t_1$]

5.8.2.5. WITHIN

The WITHIN operator defines the *validity* of an event. It is a temporal limitation on any conjunction (unordered or ordered) of two events, e_1 and e_2. It specifies a maximum duration between the first event occurrence and the second event occurrence. A WITHIN operator could be declared as follows:

e_1 AND e_2 WITHIN x HOURS/MINUTES/SECONDS

e_1 PRECEDES e_2 WITHIN x HOURS/MINUTES/SECONDS

An ESL example could be:

UPDATE student AND UPDATE StudentUnit WITHIN 24 HOURS

Formally, e_1 AND e_2 WITHIN T is expressed as:

i. [non-temporal condition (e_1, e_2)], e_{eval}=rule evaluation(r)

ii. event(e_1, t_1), event(e_2, t_2), event(e_{eval}, t_{eval})

iii. [t_2-t_1 < = T, next(t_2, t_{eval})] OR [t_1-t_2 < = T, next(t_1, t_{eval})]

5.8.2.6. BETWEEN

The BETWEEN operator defines a constraint of occurrence of an event. It is a temporal conjunction to any declared event. An event e_2 is constrained to occur between the events e_1 and e_3. A BETWEEN operator could be declared as:

UPDATE student BETWEEN 9:00-5:00

Formally, e_2 BETWEEN e_1-e_3 can be expressed as:

i. [non-temporal condition (e_1, e_2)], e_{eval}=rule evaluation(r)

ii. event(e_1, t_1), event(e_2, t_2), event(e_3, t_3), event(e_{eval}, t_{eval})

iii. ($t_1 < t_2 < t_3$) $\wedge$ next(t_3, t_{eval})

5.8.2.7. NOT

The unary negation operator may only be declared within a closed interval. The interval can be seen as being bounded by two events, which may be temporal events but need not be. The rule will be evaluated whenever e_1 occurs before e_3, and e_2 has not occurred between these two events. A NOT operator could be the declared as:

NOT e_2 BETWEEN e_1 - e_3

An ESL example could be:

NOT UPDATE student BETWEEN 12:00-13:00

Formally, NOT e_2 BETWEEN e_1-e_3 can be expressed as:

i. [non-temporal condition (e_1, e_2)], e_{eval}=rule evaluation(r)

ii. event(e_1, t_1), event(e_2, t_2), event(e_3, t_3), event(e_{eval}, t_{eval})

iii. $(\neg(t_1 < t_2 < t_3) \wedge (t_1 < t_2)) \wedge next(t_3, t_{eval})$

5.8.2.8. EVERY

The EVERY operator defines the *periodic repetition* of a temporal event. The event is continually raised after the same period from a reference point, which is assumed to be the current time. It may be declared as follows:

EVERY x HOURS/MINUTES/SECONDS

For example:

EVERY 24 HOURS

Formally, the current time t plus a period T, i.e. t + EVERY T may be expressed as:

i. e_{eval}=rule evaluation(r)

ii. event(e_{eval}, t_{eval})

iii. $(t' = t + nT) \wedge next(t', t_{eval})$

5.8.2.9. DELAY

The DELAY operator defines a relative period, from the current time, after which an event will be raised. It may be declared as follows:

DELAY x HOURS/MINUTES/SECONDS

For example:

DELAY 240 MINUTES

Formally, t + DELAY T may be expressed as:

i. e_{eval}=rule evaluation(r)

ii. event(e_{eval}, t_{eval})

iii. (t'=t + T) $\wedge$ next(t', t_{eval})

5.9. Event Parameters

In some cases it would be useful to be able to reference the object that raised a given event. For instance if aircraft movement has been detected by the radar, which has raised an event, the system will need to know the actual aircraft that caused the event. Different parts of the rule may need to reference the object that raised the non-temporal event, i.e. the condition clause or the action clause. This can be achieved by referencing the position in the event specification clause i.e. using an event parameter, using the keyword **OBJECT***n*, where *n* is replaced by the non-temporal event number. For example, in the following ESL clause

READ student

If the condition clause wanted to reference the raising object it would use OBJECT1 since the student class is the first mentioned class (it is the only class in this example). During the condition evaluation, the OBJECT1 keyword would be replaced by the ID of the actual student object, that raised the read event. Similarly, in the following example

UPDATE student BETWEEN 16/3/94-15/5/94 OR DELETE student

In the above event clause there is an implicit conjunction between the first internal event, UPDATE student, and the following temporal event, BETWEEN 16/3/94-15/4/94. The internal event, DELETE student, is actually the third event in the clause but only the second non-temporal event.

5.10. Condition Specification

REFLEX has been designed with flexibility of use in mind. To this end, as mentioned before, the condition evaluation clause for a rule may take one of four forms as discussed below:

- *TRUE* i.e. an empty condition that equates to TRUE, and results in an Event-Action pairing. This is suitable for some rules which do not need to test the internal state of the database, and simply execute some task on the occurrence of an event. For example, to initiate a backup of the database a rule could be declared as:

```
ON      EVERY DAY AT TIME 5.30PM
IF      NULL
THEN    CALL BACKUP
```

- *external* condition module using the CALL keyword. The result would be a boolean. For example, if one wanted to determine the external climate, an external module similar to the following could be called:

  call isItRaining()

  Returning a result of TRUE or FALSE. This would be particularly useful, if the internal state of the database is not required, since the external call would obviate the unnecessary update to the database simply to test the external state.

- REFLEX's high level *Object SQL* dialect. An example could be as follows:

```
SELECT    inches
FROM      rainfall
WHERE     DATE = CURRENT AND
          TIME = CURRENT AND
          inches > 0;
```

If the condition is satisfied, the SELECT returns a non-null result. This form of the condition declaration is the most portable, as REFLEX provides this for each platform, and it is close to an industry standard way of interacting with all types of databases.

- proprietary language of the host DBMS.
  REFLEX allows the user to enter queries in the native language of the host database. This allows for fast query results as the host database user may generally have greater knowledge of the host environment and thus is able to declare optimal queries.

REFLEX maps the Object SQL to the proprietary language. An application designer thus has the flexibility to write the clause in either form. The rule's condition clause is compiled, as with the other clauses, either at creation time or on modification.

5.11. Action Specification

The EECA knowledge model implemented in REFLEX allows for multiple Action and Fail Action specifications. These are a superset of those allowed for the Condition specification. The specifications for the Action and Fail Action clauses are identical, the form selected depending on whether the condition clause was satisfied or not. For convenience both Action and Fail Action will be referred to as the Action clause for the remainder of this section. There must be at least one Action specified, the forms of which are as follows:

- *external* execution module using the CALL keyword. For example, In an Air Traffic Control System, an external call could be made to open a dialogue window on the operators screen to either Alert i.e. some dangerous situation, or prompt the capture of data about a given

situation i.e. a new aircraft has entered the airspace

```
CALL        AlertOperator OBJECT1
```

The external AlertOperator function is passed the name of the object (aircraft) that raised the alarm.

Hence the database would be initiating external activities.

- REFLEX's high level *Object SQL* dialect is as described for the condition clause, but with further extensions to allow for the insertion of new objects into the tables or class instance space. An example could be as follows:

```
INSERT INTO      reorder_log (item_id)
VALUES           (OBJECT1)
```

i.e. if the update of a certain stock item caused its quantity-on-hand to fall below a certain level, enter the particular item into the reorder log.

This form of the action declaration is again the most portable.

- proprietary language of the host DBMS.

As with the condition specification, REFLEX allows the user to enter queries in the native language of the host database, allowing for faster more optimal query results.

5.12. Example EECA Rules

Example EECA rules could be as follow:

- Air Traffic Control System

  Consider a rule to test whether an aircraft which has changed its position is in danger by moving to close to another aircraft. The rule is brought into context after an update to the database by a simple/primitive event. An OSQL query tries to determine whether the aircraft in question is in the vicinity of another aircraft. If so, the operator is alerted, and a log entry made.

```
E     AFTER UPDATE aircraft
C     SELECT      a.Name()
      FROM        aircraft a, aircraft b
      WHERE       a.Name() = OBJECT1
            AND   (a.CurX - b.CurX) BETWEEN -5 AND 5
            AND   (a.CurY- b.CurY) BETWEEN -5 AND 5
            AND   (a.CurZ - b.CurZ) BETWEEN -5 AND 5;
EC    immediate
A     (AlertOperator OBJECT1; immediate; independent)
      (INSERT ON log a.itemID, XYZ; decoupled; independent)
FA    NULL
```

- Stock Control

  In this scenario, if an item is sold, after the database has been updated a test is made to determine whether the quantity on hand is less than a minimum threshold. If so, a reorder item is created on the reorder table.

```
E     AFTER UPDATE item
C     SELECT      a.Name
      FROM        item a
      WHERE       a.Name = OBJECT1
```

```
                    AND     a.QtyOnHand < a.MinQty;
EC      deferred
A       (INSERT ON reorderItem a.itemID, a.ReorderQty; decoupled; independent)
FA      NULL
```

5.13. Summary

This chapter has introduced REFLEX's EECA knowledge model, which with its multiple action and fail-action clauses and its associated extension of coupling modes, is significant because it alleviates the problems caused by situation redundancy i.e. replication of rules simply because they have the same event and condition clauses.

The section on coupling modes highlighted several problems of application semantics caused by the EECA polyform, mainly the dependency issue. This has been resolved by introducing the action clause tuple that includes a dependency flag for each individual action or fail-action clause. The designer of the application system is given the choice as to what level of transaction dependency is required for a given application.

It is believed that the EECA knowledge model proposed does in fact allow the declaration of the knowledge within the active database system to be both semantically concise and obvious as to its intention. The model also allows for a more efficient evaluation and operation of the overall active database system.

The representation of both rules and events as first-class objects were described together with the rationale for the choice of their representation method i.e. uniformity of representation, scope for optimisation, dynamic definability.

The complex event specification method employed by REFLEX was described in relation to related work and its semantics. The issues of event chronology, interval

logic and validity were illustrated, and lead to the English ESL. English ESL provides similar complex event specification facilities to systems such as Ode [Gehani 92a] and SAMOS [Gatziu 93], but unlike the other systems the semantics associated with the English ESL have been critically specified using a modified form of the temporal logic of Knight [Knight 92b].

The chapter concluded with the semantics of both the condition and action specifications, and how they provide flexibility to the designer of an active application by providing many forms of specification.

Chapter 6

Design Architecture and Implementation

This chapter overviews the architecture of REFLEX, and later discusses its implementation. The portability and adaptiveness of the REFLEX extension to a given DBMS, is examined and lessons learned by its implementation on two different platforms namely ONTOS (SUN Solaris, X11) and POET (PC, Windows). The adaptability of the model are reported within the chapter.

6.1. Introduction

REFLEX provides active functionality for a host object DBMS by introducing some new classes. The most important of these classes are as follows:

- active object

  all application classes which require the notion of activity must ultimately be derived from this class
- rule

  which encodes the EECA knowledge model
- event

  events are represented as objects which maintain links to affected rules
- knowledge manager

a central scheduler of the knowledge execution within the database.

REFLEX has been engineered to adapt to different host DBMSs. This ability for a general extension to be adaptive is investigated within this chapter.

As reported by Chakravarthy et al. [Chakravarthy 89], the HiPAC active database has to manage the component parts of its system i.e. the objects, transactions and rules, and hence it supports an object manager, transaction manager and rule manager. This is not the case with REFLEX as it does not need to know how the objects themselves are managed since this task is left to the underlying host DBMS. The system is composed of layers, each of which have defined interfaces which allow low level services such as the searching and retrieving of data, to be simply requested from the host DBMS. Essentially embodying the modern day Software Engineering paradigm of software component libraries and their use.

This chapter is organized as follows. As described in chapter four, the underlying host databases are object-oriented, these are discussed in section 6.2 allowing the architecture and implementation decisions to be understood. An overview of the architecture is then presented in section 6.3, followed by detailed descriptions of the components of the model. Section 6.5 introduces the distribution features of the model, which then leads to the section which discusses performance issues. The user interface, Vis, in introduced in section 6.7. The portability and adaptability of the model are demonstrated in section 6.8.

6.2. Object Databases

Object oriented database environments require that the modeled objects exist or persist after the processes that created them. This is the task of a persistent store or minimal database system, as discussed in chapter two.

REFLEX has been designed as an extension to an object-oriented host DBMS. It has been implemented upon two such object-oriented database systems ONTOS [ONTOS 91] and POET [GWB 92], these DBMSs will be briefly discussed in turn, emphasizing their differences.

6.2.1. ONTOS

ONTOS [ONTOS 91] provides persistence and other data management facilities for C++ objects [Stroustrap 86]. It is a relatively mature distributed client-server object database that distributes the database around a network of homogeneous workstations, figure 6.1. It has all of the object modelling tools expected of an object-oriented DBMS i.e. inheritance, polymorphism, address translation, global naming schemes, advanced

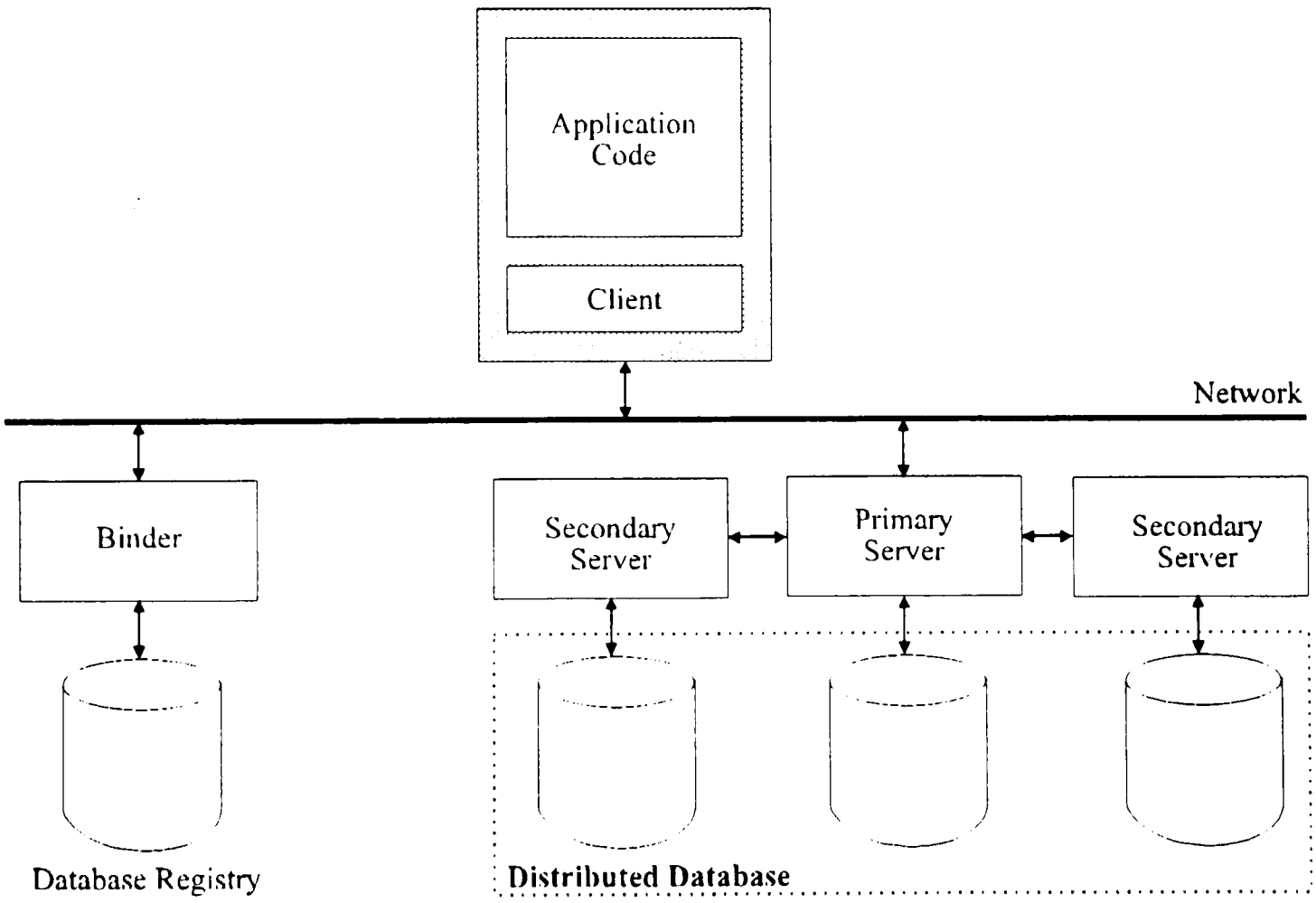


Figure 6.1 ONTOS DB distributed database

transaction processing, concurrency control, distribution, and custom storage manager facilities. Unfortunately, being a new type of database technology user interaction is

restricted. Access is gained by calling its libraries by programming in C++, although some of the later tools are graphical and claim to be 4GLs, they are still essentially 'screen painters'.

ONTOS provides persistence for application objects by means of a base class, OC_Object, which all objects that require persistence must inherit from. Various aggregation (or collection) classes are provided such as arrays, dictionaries, lists and sets which also inherit from OC_Object. All ONTOS classes ultimately inherit from its root class, OC_Entity, as can be seen in figure 6.2.

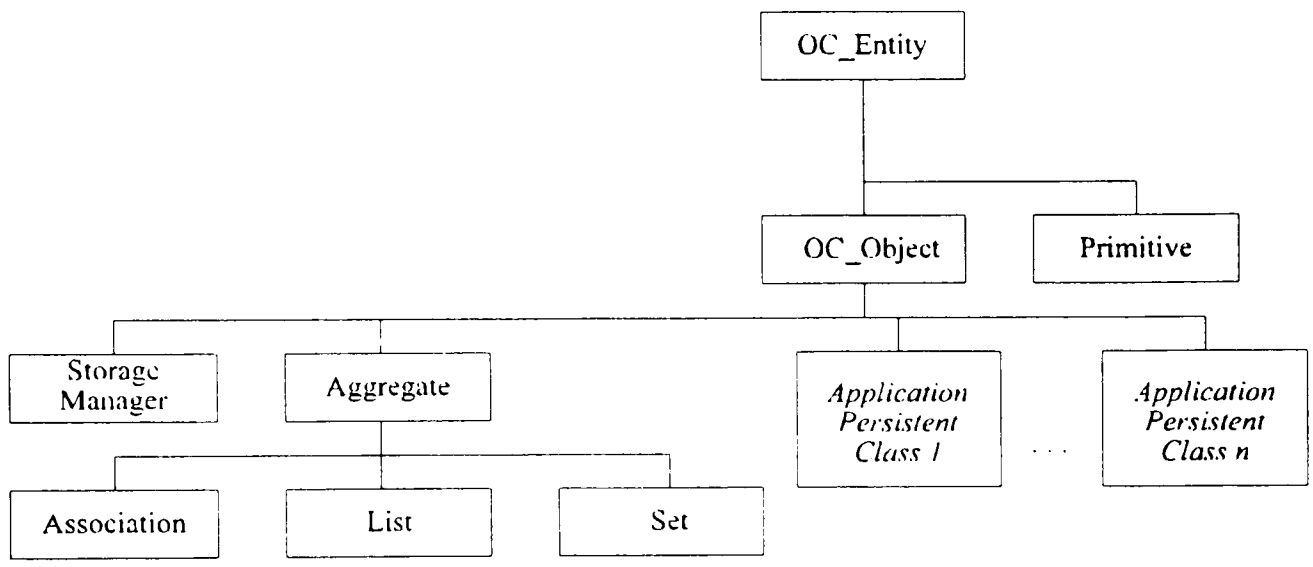


Figure 6.2 ONTOS base class hierarchy

The requirement for a persistent object-oriented environment is that objects must be saved to disk and retrieved at a later time, in their exact same state. For a language such as SmallTalk [Goldberg 81], which is object-oriented in the pure sense of the word since it treats everything as an object, this process is simple although clumsy as it saves its entire environment image to disk. The entire environment is reloaded into memory the next time it is required. This simplicity can be afforded because SmallTalk is an interpreted language. C++ is a hybrid object-oriented language where object

extensions have been added to the compiled C programming language [Kernighan 78]. Unlike Smalltalk, C++ is essentially static, i.e. all of the *type* information about application objects is processed at compile time and is not available at run-time. This type definition is required in persistent environments where the object may be retrieved at a later date, by a different process than the one that originally created the object. If the type definition is not available at run-time the retrieving process would not be able to distinguish the member properties or methods, for example if Joe is of type Person, without the definition of Person i.e. as having the following attributes name, sex, ..., NI number, could not be loaded. ONTOS does, however, provide this information. This is accomplished by registering type information into a schema database which can be interrogated at run-time.

6.2.2. POET

POET [GWB 92], which stands for Persistent Objects and Extended database Technology, like ONTOS provides persistent storage for objects. It is not, however, as mature as ONTOS but does try to provide many of the same features. For this research a stand-alone version of POET was used, there are however professional versions which offer client-server functionality similar to that provided by ONTOS. The standalone version does have some of the features such as collections (in the forms of sets), transactions, references etc. POET is available on many platforms such as UNIX (Sun Solaris), Macintosh, and Windows (3.11 and NT). For the purpose of the research, a simple prototype was required to show that the system was indeed portable and adaptive, so the Windows 3.11 platform was selected primarily because of cost.

POET like ONTOS, provides C++ class information at run-time. This is achieved by registering the class definitions into a database, which then can be used at run-time.

For a POET application objects to become persistent, they must be declared using the

keyword *persistent* e.g. for a Person class the following declaration could be used:

```
persistent class Person {
private:
        . . .
public:
        . . .
};
```

The following sections introduce the REFLEX architecture and how the modules logically work together. Throughout this chapter the ONTOS implementation will be used to demonstrate the various aspects of the model. The POET implementation will be referred to in section 6.8, which demonstrates the portability and adaptiveness of the model.

6.3. REFLEX Architecture

REFLEX, as an active database extension, deals with explicit knowledge in the form of rules. The rules have event specifications, condition specifications and triggered action declarations. In order to process these items REFLEX, like other active databases such as HiPAC, has knowledge, event, transaction and execution models.

The above models are embodied in design of the REFLEX architecture which has the following major logical components, figure 6.3:

- Knowledge Management Kernel (KMK)
- Event Manager (EM)
- Knowledge Selection Module (KSM)
- Condition Evaluation Module (CEM)

- Execution Supervisor (ES).

As well as the above mentioned components REFLEX has a module, the Transparent Interface Manager (TIM), which interfaces REFLEX to any given host DBMS, and mainly affords the flexibility and adaptability features of REFLEX. This module is novel to the genre since other prototype active DBMS (HiPAC, StarBurst, POSTGRES, ADAM) are closely linked to their underlying DBMS. REFLEX, similar to HiPAC and ADAM, is designed and built as an object-oriented system.

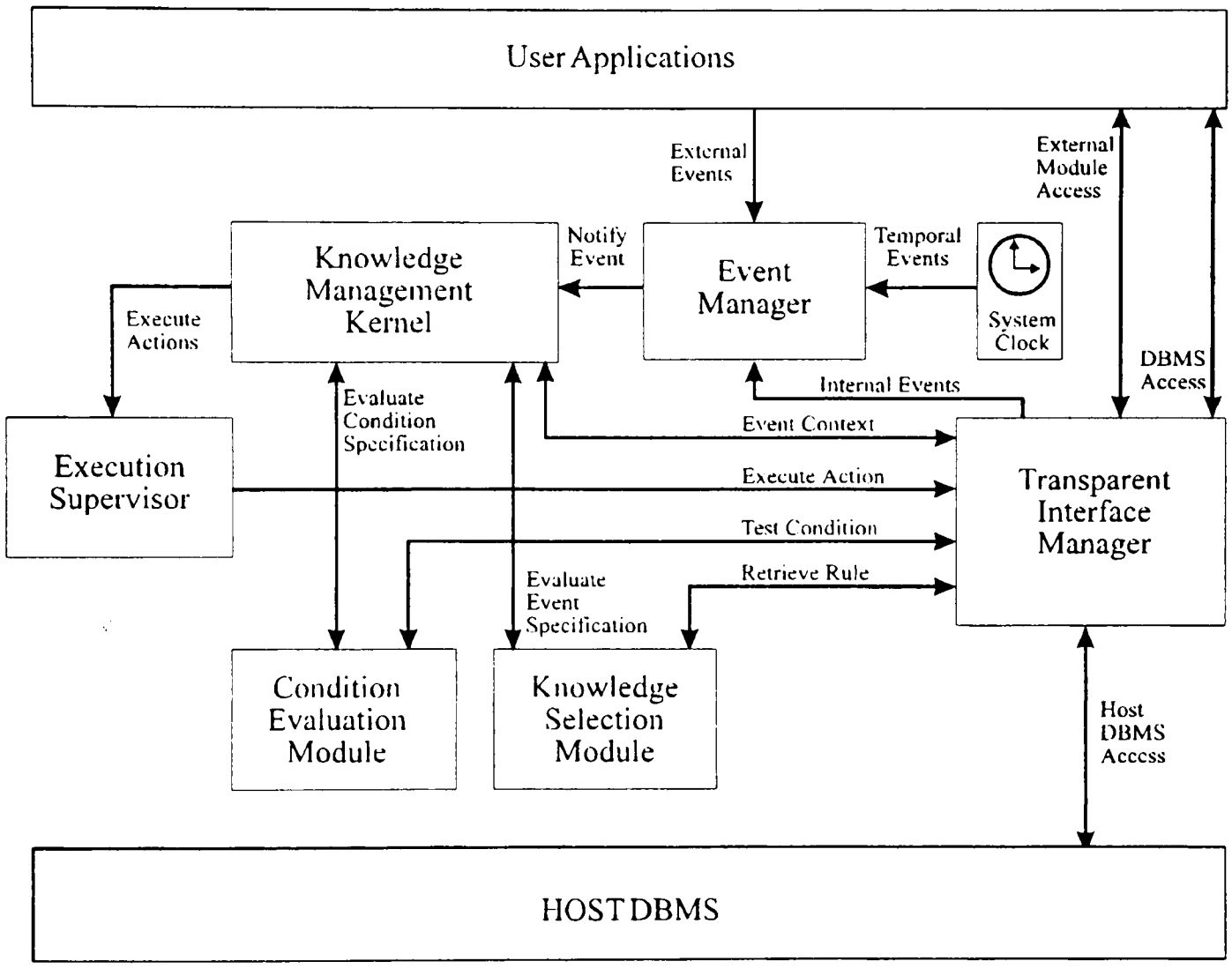


Figure 6.3 REFLEX Architecture

As can be seen from figure 6.3, the events are raised and signalled to the Event Manager from three sources (i.) internal events by the Transparent Interface Manager (ii.) external events by the application programs and (iii.) temporal events by the

system clock. The Event Manager is responsible for the logging of the events and their notification to the Knowledge Management Kernel, which evaluates whether the event affects any rules. If the event affects any rules, the rules in question are passed to the Knowledge Selection Module, which evaluates whether the rule's event specification clause has been satisfied. If it has been satisfied, then the rule is returned to the KMK ready for its condition clause to be tested. The KMK passes the rule to the Condition Evaluation Module which tests the condition clause. If the clause is satisfied, the CEM returns the rule with a status of 'fireable'. The KMK then passes the rule to the Execution Supervisor, which then executes the actions.

The component modules are described in the following section.

6.4. Components of the Model

6.4.1. Transparent Interface Manager (TIM)

For a given database to become active, one of the most important features is that the occurring events must be detected. It is the TIM that allows *internal* host database events to be trapped and signalled to the Event Manager. For this to occur access to the database must go through the TIM. Internal events that the TIM signals are database operations such as *read*s, *write*s, *update*s and *delete*s and events that support transaction atomicity such as *transaction start*, *transaction commit* and *transaction abort*s etc.

Database operations are detected by the provision of an active object class. This class inherits from a base class provided by the host DBMS, figure 6.4, which allows an object to persist. Using the object-oriented modelling feature of *polymorphism*[1], the

[1]Where the same operation may behave differently on different classes [Rumbaugh 91].

active object class provides *over-ridden*[2] host DBMS access functions, such as Read, Update, Write, etc. These over-ridden functions when called, provide signals to the Event Manager as well as passing the original message through to the host base function.

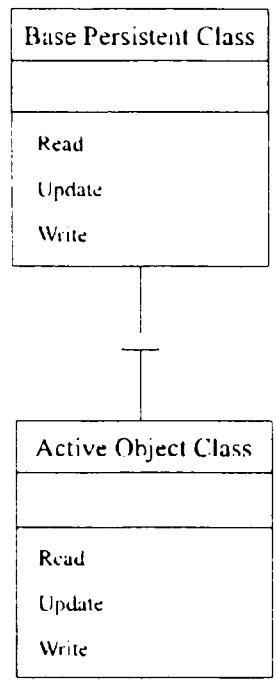


Figure 6.4 Active Object Class

Transactions in a host DBMS are provided either by free functions i.e. library functions which are not part of any class e.g. *transactionStart()* as in ONTOS [ONTOS 91], or by transaction classes as in for example, POET [GWB 92] and ObjectStore [ObjectDesign 93].

If free functions are used, the underlying database's transaction manager can be harnessed using *wrappers* [Dittrich 91], where its interface is encapsulated by special wrapper functions which inform the Event Manager that a transaction based event has taken place, to allow the detection of transaction events and also to allow the creation of nested and sibling transactions.

---

[2]A subclass may override a superclass feature by defining a feature with the same name [Rumbaugh 91].

```
REF_transactionStart(args...) {
        em.raise(before, transactionStart, ...);
        OC_transactionStart(args...);
        em.raise(after, transactionStart, ...);
}
```

Figure 6.5 REFLEX example transaction event raise wrapper

The REFLEX Knowledge Model uses the notion of intervals for the occurrence of internal events. Since an interval dictates a point in time just before or after an internal event, an event is raised both before and after the actual host DBMS function call. Figure 6.5. illustrates this principle with some example code.

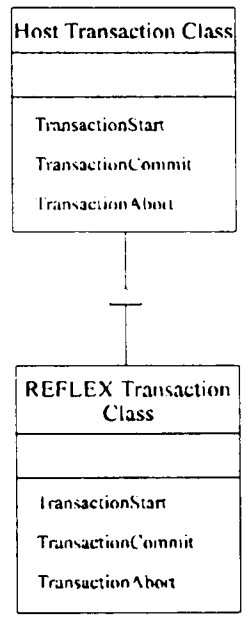


Figure 6.6 Signal Generating Transaction Class

If the transaction scheme for the host DBMS is class based then active transaction classes are sub-classed in a similar fashion to the Active Object Class, but from a base transaction class, figure 6.6. The base transaction methods are over-ridden in the new transaction class, to provide an event signal before passing the message through to the actual base transaction method.

6.4.1.1. The Active Object Class

Access to the active features is serviced by the provision of an active object class, *AObject*. If an application class is required to be able to activate the system, it must ultimately inherit from AObject. This class inherits from ONTOS's OC_Object class, as can be seen in figure 6.7 and in the C++ definition code fragment figure 6.8.

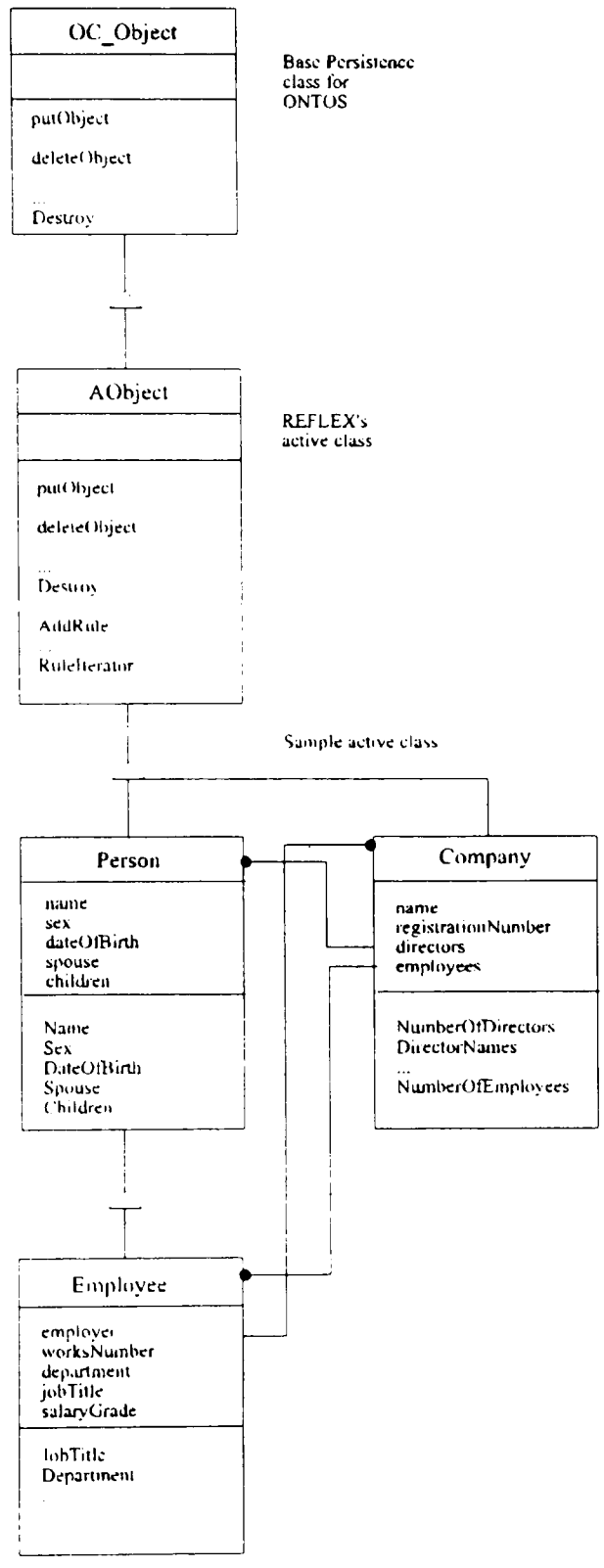


Figure 6.7 Active Signalling Inheritance Hierarchy for ONTOS

```
class AObject : public OC_Object {          // subclass from ONTOS's base class

private:
        char*                   AObjectName;
        Reference               ActiveRules;     // Rules on Class
        Reference               ExemptRules;     // Rules the object is exempt from

public:
        // Constructors/Destructors

        AObject(char* name=0);
        AObject( APL * theAPL );
        ~AObject();

        // DBMS functions
        virtual void            Destroy( Boolean aborted=FALSE);
        virtual Type*           getDirectType();

        virtual void            putObject( Boolean deallocate=FALSE );
        virtual void            putClosure( Boolean deallocate=FALSE );
        virtual void            deleteObject( Boolean deallocate=TRUE );
        virtual void            lockObject( LockType );

        // Methods for Rules Dictionary - Housekeeping
        void                    AddRule(Rule*, int fromRule = FALSE);
        void                    RemoveRule(Rule*, int fromRule = FALSE);
        int                     HowManyRules();
        Rule*                   FindRule(char*);
        void                    deleteRuleLinks();
        RfBoolean               HasRule(Rule*);
        AggregateIterator*      RuleIterator();

        // Methods for Exempt Rules Dictionary
                                . . .

        // Accessors
        void                    CallingRule(Rule*);
        virtual  void           Name(char* newName);
        virtual char*           Name();
};
```

Figure 6.8 AObject Definition Code

6.4.1.2. Transaction Free Functions

To manage transaction points ONTOS models transactions as library free functions. Since REFLEX must *adapt* to the same mode of operation as the host DBMS, ONTOS, it also models transaction calls as free functions. As can be seen in figure 6.9, a

REFLEX function REF_transactionStart(...) wraps around the host OC_transactionStart() free function call.

```
void REF_transactionStart (
        XAType                  Orig_RWConflict,        // conflict
        XAConflictResponse      Orig_waitOnConflict,    // conflict resolution
        char*                   str,                    // name
        BFP                     Orig_buf )              // buffering
{
        // Call Event Handler
        EventDetector     evdet;

        evdet.eventRaiseTrans     (START,BEFORE,"Raising event from BEFORE TransStart");


        // Call original ONTOS function
        OC_transactionStart       (Orig_RWConflict, Orig_waitOnConflict,
                                   str, Orig_buf );

        evdet.eventRaiseTrans(START,AFTER,"Raising event from AFTER TransStart");

};
```

Figure 6.9 REFLEX transaction function call for the ONTOS DBMS

This method allows the event signal to be generated both before and after the actual event.

6.4.2. Event Manager (EM)

As events are raised they are signalled to the Event Manager which is responsible for both their recording and notification within the system.

As stated earlier, REFLEX supports composite events for which the component events occur at different points in time. Each occurrence of an event must be recorded i.e. a *chronology* or history needs to be maintained.

When an event is detected, in order to satisfy the requirement that the event chronology must be maintained, the EM logs the occurrence of the event in the *temporal log*. The EM then informs the Knowledge Management Kernel that an event has occurred.

6.4.2.1 Event Monitoring

Detection of the different event types (internal, user-defined and temporal), must occur in order for the system to react. This is the responsibility of the Event Manager, figure 6.10.

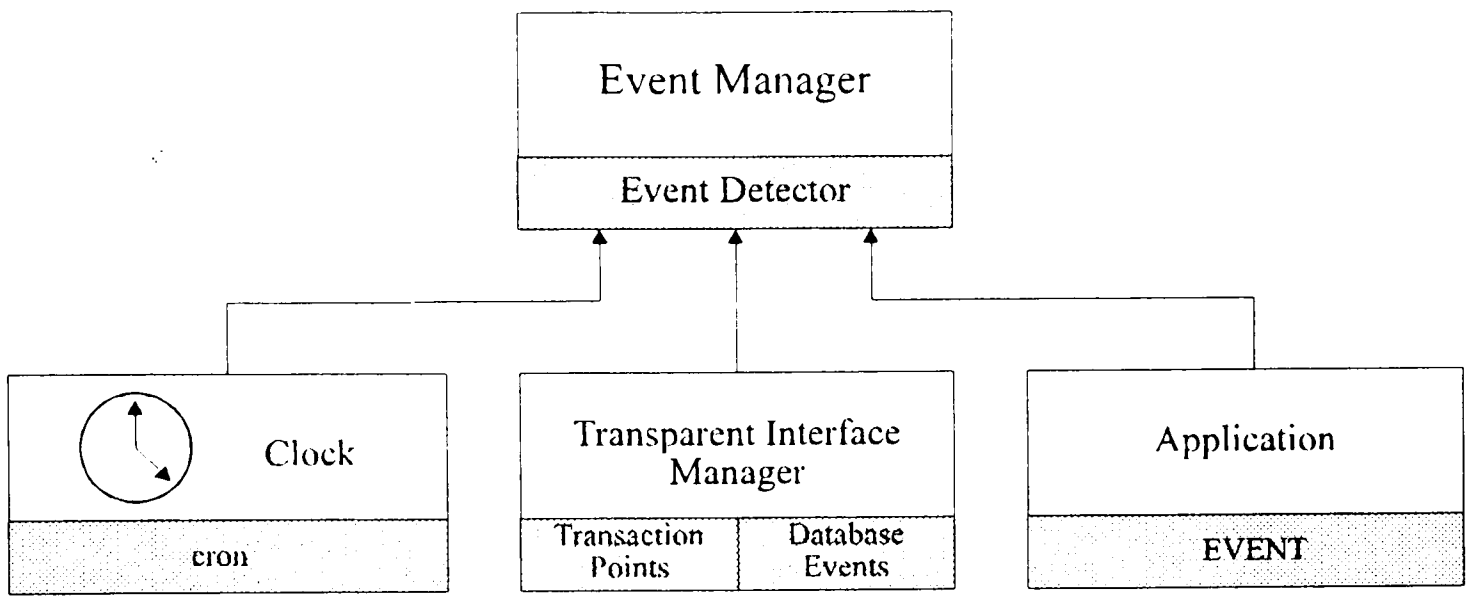


Figure 6.10 Event Signal Generators

Primitive internal database operations may be detected by building *layers* around the database operations, which raise the event if called. This approach has been followed for both event generator classes and free library functions, as was discussed previously in the section on the Transparent Interface Manager.

External application generated events are detected by the application program explicitly calling the event detector's raise signal. This method does not cause any noticeable overhead as it does not require the application to modify the database artificially, simply to raise an event.

Temporal events may be detected in a variety of ways. Depending on the platform, certain clock based facilities are available, for example the unix *cron* facility may be used which can be set to notify at some particular time, or periodically, as used by SAMOS [Gatziu 93]. This approach is obviously not portable, cron being available on unix only, and so is not used with REFLEX. Instead, to detect temporal events, the Event detection module itself has the temporal event generator built in.

```
void EventDetector::eventRaiseDB(EventType et, AObject *aobj,
                    Interval inter, char* theEventDesc) {
    // Retrieve event ---------------------------------------------
    EventObject    *evnt = getIntDBEvent(et);
    // test if event affects any rules
    if (!(evnt->HowManyRules())) return;

    // Get local occurring time
    struct tm localTime;
    struct tm *plocTime = &localTime;
    time_t clock;
    time(&clock);
    plocTime = localtime(&clock);

    // Stamp on Temporal Log
    char   NoStr[20];
    NoStr[0] = '\0';
    ADBGetLogNo(NoStr);    // Get new log number

    TemporalLog    *tLog = new TemporalLog(evnt,INTERNAL, aobj, NoStr);

    tLog->setTime1(plocTime);
    tLog->intDBInt(inter);
#ifdef ONTOS
    tLog->putObject();
#elif POET
    tLog->Store();
#endif
    // Inform Global KMK
#ifdef ONTOS
    RM = ADBGetRM(WriteLock);
#endif
    RM->knowledgeScheduler(evnt,tLog);
} // eventRaiseDB
```

Figure 6.11 Event Manager - internal event raise code segment

6.4.2.2. Temporal Log

The Event Manager, on the occurrence/detection of an event, quickly ascertains whether the event applies to any of the rules within the active application domain. If the event affects *at least one* rule, then its occurrence is recorded on the temporal log.

This process is illustrated by the code fragment, figure 6.11, which shows the raise signal for an internal event, recording the event onto the temporal log and then informing the KMK of its occurrence.

The log records the occurring event, the object that caused the event, and the time of its detection. It is assumed that the time of occurrence of an event and its detection are the same.

6.4.3. Knowledge Management Kernel (KMK)

Most active database (or event driven) systems, for example HiPAC [McCarthy 89] and ADAM [Diaz 91b], require a central control module. In REFLEX this is the responsibility of the Knowledge Management Kernel.

The KMK acts as the nucleus of the REFLEX architecture. Its major tasks are divided between being a command dispatcher to the other modules, and more importantly, as a rule evaluation scheduler.

REFLEX is designed to be used in many areas i.e. administrative computing for example, Student Record Systems, payroll etc., attention has also been given to the real-time application domain. Hence, it must respond very quickly to be able to influence the environment of which it is a part; e.g. in process control applications. For this reason many of the modules have been designed to operate autonomously and concurrently. The KMK is responsible for scheduling the cooperating modules which frequently act on the same data space. The interfaces between the KMK and the five modules are described briefly below and in more detail later in their respective sections.

6.4.3.1. EM-KMK-KSM Interface

An event generator signals the occurrence of an event to the EM. If the event affects any rules, this is ascertained since the signalling event object maintains a list of the rules it may affect, the EM records the event occurrence and notifies the KMK of the occurrence.

For each rule that appears in the event's list of affected rules a Knowledge Selection Module (KSM) is instantiated, and a rule passed to it. The KSM evaluates the rule's event clause and returns a status of '*in-context*' for the rule if the clause was satisfied.

6.4.3.2. KMK-CEM-ES Interface

If a rule is returned with a status of 'in-context' from the KSM, its condition clause may then be tested. The KMK submits the rule to a Condition Evaluation Module (CEM) which tests its condition clause and if satisfied, returns a status of '*fireable*'.

On return from the CEM, if the rule's condition clause has been satisfied (rule is *fireable*), it is then passed to the Execution Supervisor (ES). The ES then executes each of the action clauses for a fireable rule, subject to their condition-action coupling modes. If the condition clause failed, the EECA knowledge model supports fail actions, these would be executed again subject to condition-fail action coupling modes. If there were no fail-actions the rule is discarded from working memory and no further action need be taken.

6.4.3.3. KSM and CEM Concurrency

Normally, the condition clause for a rule is not evaluated until the rule has a state of *in-context* i.e. until the event clause has been satisfied. REFLEX provides a novel concurrency method which depending on whether a rule has the top-most trap priority set, the KMK may simultaneously dispatch it to the KSM and Condition Evaluation

(CEM) modules, figure 6.12. This design strategy ensures parallelism and faster availability of the two results. If both results are true, the rule's action clause may be passed to the Execution Supervisor immediately. However, if the KSM returns an unsatisfiable rule, the CEM is preempted, and the result discarded. If the CEM returns its result first, this is written onto the *pipeline* log ready for use.

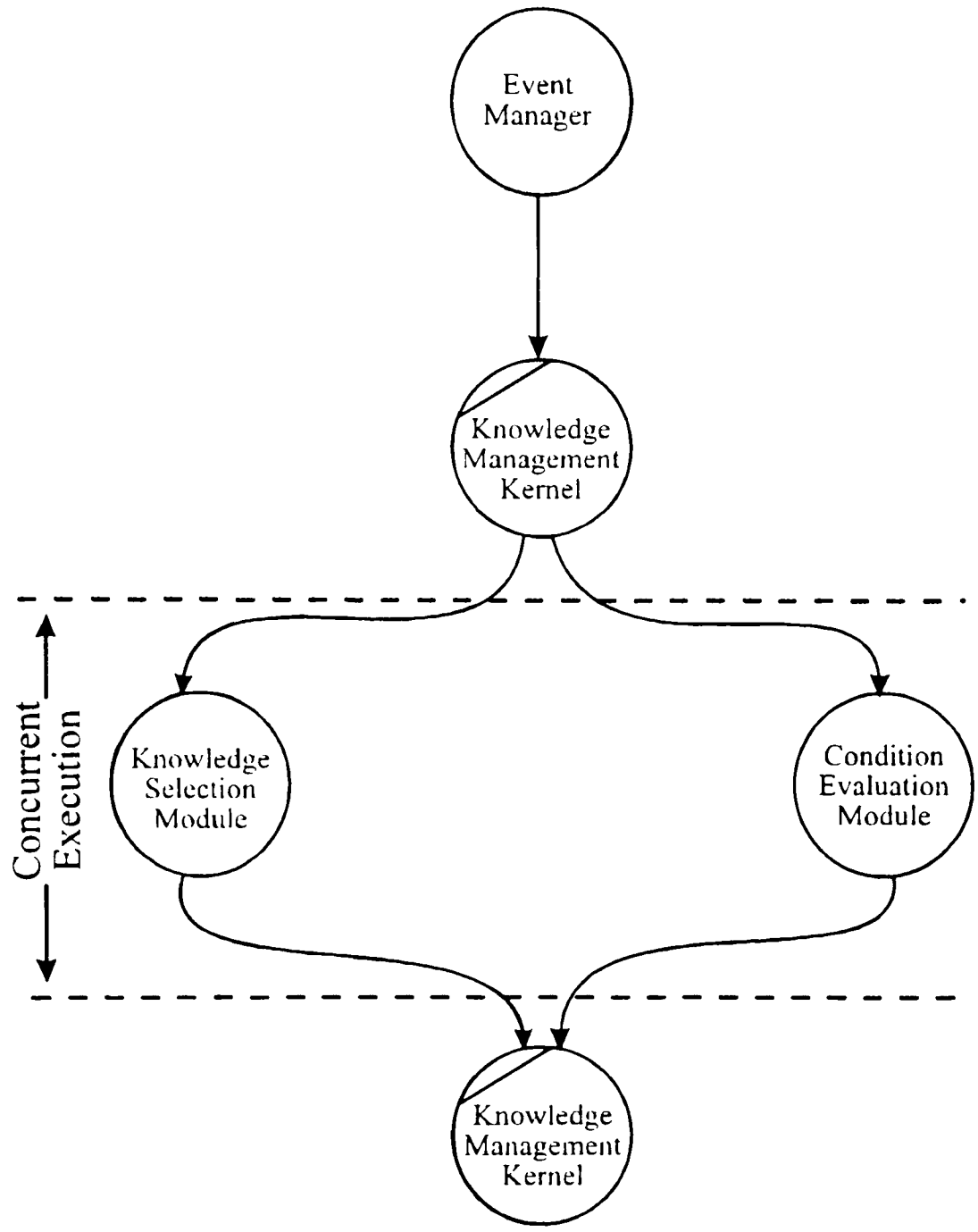


Figure 6.12 Concurrent Execution KSM & CEM

This feature is desirable in *critical real-time* situations. It may only take place at the application designers discretion by setting the high *trap* priority for any critical rules. However this feature may not always be desirable as efficiency of the database may be adversely affected. But by using the parallelism only for critical, high-priority rules it

could improve the response time of the overall system without overloading the system by causing unnecessary condition clause evaluation. However, this, as indicated above, is an efficiency decision made by the designer.

This novel concurrency approach enables the construction and implementation of high performance systems, capable of modelling real-time critical applications.

6.4.4. Knowledge Selection Module (KSM)

On being called by the KMK, the KSM tests the event clause of a given rule. If a rule has a simple (i.e. single) event of the kind that occurred, and it is satisfied, it is returned to the KMK with a status of *'in context'*.

```
static PartCompEventSpec     *pces;     // For part compiled clauses

int         KnowlSel::testEventSpec(Rule* rule, TemporalLog* tLog) {
            int         noClauses = rule->numSpecParts();
            int         index = 0;
            int         retVal = FALSE;
            int         partSatisfied = FALSE;

            pces = rule->OwnerOfPCES();
            if (!pces)  // New event, set up PartCompiledEventSpec
                        pces = new PartCompEventSpec(rule, noClauses);
            struct tm dectionTime;
            getSysTime(&dectionTime); // set the time to current system time
            do {
                        CompiledClause *cl;
                        cl  = pces->getCompiledClause(index);
                        if (!cl)  cl  = new CompiledClause(FALSE, &dectionTime);
                        if (!cl->Satisfied()) { // Current cluase is not satisfied
                                    Clause *cl = rule->ruleClause(index);
                                    retVal = testSingleEvent(cl, tLog, &dectionTime);
                                    if (noClauses == 1)             // Simple event
                                                return retVal;
                                    if (retVal == TRUE) {           // Part Satisfied Specification
                                                printf("\nCOMPLEX EVENT, clause %d satisfied", index);
                                                pces->memberClause(index, &dectionTime, TRUE);
                                                partSatisfied = TRUE;
                                    } else
                                    if (strcmp(cl->keyWord(),"WITHIN")==0) {
                                                pces->memberClause(index, &dectionTime, VALIDATION);
                                                pces->validitySeconds(cl->validitySeconds());
                                    } else      pces->memberClause(index, &dectionTime, FALSE);
                        } // IF false
            } while ((++index < noClauses) );
            if (partSatisfied == TRUE) {   // test if specification has been satisfied
                        int ret;
                        char        buf[200];
                        struct tm OccTime; initTime(&OccTime);
                        strcpy(buf, reverseWords(rule->rpnStr()));
                        ret = RPNexpressionEval(rule->rpnStr());
                        if (ret == TRUE) {
                                    printf("\n\n\nComplex Event Returned TRUE");
                                    pces->deleteObject(); // pces not required any more
                                    return ret;
                        } else {    printf("\nComplex Event Returned FALSE");
                                    pces->putObject();
                        }
            }
            return FALSE;
} // testEventSpec
```

Figure 6.13 KSM - testEventSpec

If the rule's event clause is complex, the individual component events occur at different points in time, each of which have to be checked on occurrence. If the current part of the event specification has been satisfied, then the KSM checks the temporal log for any relevant related events that have occurred previously. The code fragment, figure 6.13, illustrates this. If related events have occurred previously, it checks if the event

specification is now completely satisfied, if satisfied the KMK is informed that the rule is '*in context*'. If the rule's event clause is only part-satisfied at this stage, the part-satisfied clause is written to the *pending* log, together with a copy of the state of rule evaluation, and kept for a given period of time, until its event clause is either later satisfied or discharged.

The complex event, specified in English ESL, is parsed and syntactically checked and part-compiled on entry. The KSM is responsible for the semantic checking and evaluation of the event specification at run-time. It must ensure that the intended logic is obeyed. The code fragment, figure 6.14, illustrates the checking between the logical operators i.e. PRECEDES and SUCCEEDS are ordered whilst AND and OR are unordered.

```
// calculate difference in operator times
timeDifference = (int) difftime(&time1,&time2);
int EventSatisfied = FALSE;
switch(tokNum) {
        case AND:       if (Operand1 && Operand2)
                                EventSatisfied = TRUE;
                        break;
        case PRECEDES:  if (Operand1 && Operand2)
                                if (strcmp(asctime(&time1), asctime(&time2))>=0)
                                        EventSatisfied = TRUE;
                        break;
        case SUCCEEDS:  if (Operand1 && Operand2)
                                if (strcmp(asctime(&time1), asctime(&time2))<0)
                                        EventSatisfied = TRUE;
                        break;
        case OR:        if (Operand1 || Operand2)
                                EventSatisfied = TRUE;
                        break;
        default:
                        EventSatisfied = FALSE;
        }
return EventSatisfied;
```

Figure 6.14 KSM - semantic testing of logical operators

As well as ensuring each part of the event specification has been satisfied and that logical semantic integrity has been preserved, the KSM must ensure that any validity constraints are adhered too. This is illustrated in the code fragment, figure 6.15.

```
int evalClause(char *str, int indexPos, struct tm* OccurranceTime, int& timeDifference) {
        int clauseNum = 0;
        printf("\nevalClause: %s at pos %d",str,indexPos);
        sscanf(str, "C%d",&clauseNum);
        printf("\n::evalClause Caluse %s is numbered %d",str,clauseNum);
        struct tm detectTime; initTime(&detectTime);
        int ret = pces->memberClause(clauseNum, &detectTime);
                if (ret == TRUE) {
                        // get time of event occurrence
                        datecpy(OccurranceTime, &detectTime);
                } else
                if (ret == VALIDATION) {
                        printf("\nVALIDATION Clause.....");
                        if (timeDifference > 0 &&
                                (timeDifference < pces->validitySeconds()))
                                ret = TRUE;
                }
        return ret;
} // evalClause
```

Figure 6.15 KSM - preserving validity constraints

If the rule specifies complex events in order for the event clause to be satisfied each individual component event, depending on the logical algebra, needs to be satisfied. For example, the statement $e_1 \wedge e_2 \wedge$ must be completely satisfied because it is a total conjunction i.e. all of the ANDs must be evaluated before the outcome of the clause can be known. But the following statement need not be completely satisfied, since it contains a disjunction, $e_1 \vee e_2$, where all of the operands need not occur as long as one of the two operands either side of the OR occur, the clause can be satisfied. REFLEX short-circuits once the outcome is known i.e. in the above example, if either of the OR operands occurred, the result would be returned immediately without testing the remainder of the clause.

6.4.5. Condition Evaluation Module

Once a rule's event clause has been satisfied, its condition predicate can be evaluated. To allow flexibility for a user, as described earlier, a number of different forms of condition declaration are supported within REFLEX. These different permutations are catered for and illustrated in the CEM, figure 6.16.

```
int     ConditionEvaluator::executeCommand(Rule* rule, TemporalLog* tLog) {
        if ((strlen(rule->conditionStr())) == 0) return TRUE;
        char mappedStr[1000];          mappedStr[0] = '\0';
        char conditionTypeStr[100];    conditionTypeStr[0] = '\0';
        int retVal = FALSE;

        strcpy(mappedStr, mapEventParameters(rule, tLog));

        // Determine if valid condition type and despatch for processing
        char commandType[1000]; commandType[0] = '\0';
        strcpy(commandType, getToken(mappedStr,0,TRUE));

        if (strcasecmp(commandType,"SELECT") == 0)
                retVal =  ConditionEvaluator::parseQuery(mappedStr);
        else
        if (strcasecmp(commandType,"CALL") == 0) {
                AppObject app;
                retVal =  app.executeCommand(mappedStr);
        } else
        if (strcasecmp(commandType,"HOST") == 0) {
                HostObject host;
                retVal =  host.executeCommand(mappedStr);
        }

        return retVal;
} // executeCommand
```

Figure CEM - Four types of condition clause

If the condition clause was specified as NULL, then the CEM simply returns a query result of TRUE. For all other cases, event parameters have to be mapped to the condition string, before being serviced. These are the names of the actual objects that caused non-temporal events to be raised, and may be part of a condition statement.

If the statement is a call to an external condition module, the modules is called and the CEM returns its result either TRUE or FALSE.

A condition in the form of the host DBMS proprietary language, is passed to it via a HostObject. For an Object SQL condition, REFLEX maps the OSQL to the proprietary language, using again a further call to HostObject. An application designer thus has the flexibility to write the clause in either form. The rule's condition clause is compiled, as with the other clauses, either at creation time or on modification.

6.4.6. Execution Supervisor

The EECA model allows for multiple action and/or fail-actions. If the condition is satisfied, the action clauses are then executed. The clause is similar to the condition clause as described above, except that there may be multiple clauses, but there must always be at least one.

```
int ExecutionModule::executeCommand(Rule* rule, TemporalLog* tLog, int action, int actionFlag) {
        AppObject           app;
        int                 exeResult = 0;
        char                raBut[300+1];
        char                *raStr, *raCA, *raDep, raStrTrunc[31];
        RuleAction          *raObj;
        AggregateIterator *raIterator = rule->RuleActionIterator(action);
        int                 rarow = 0, radel, raNoRows;

        if (!actionFlag) printf("ExecutionModule::executeCommand - FailAction! requested");
        while (raIterator->moreData()) {
                raObj = (RuleAction*) (Entity*) (*raIterator)();
                raStr = raObj->executionStr();
                printf("%s ",raStr);
                raCA = raObj->CAcouplingStr();
                raDep = raObj->dependencyStr();

                if (raStr) {
                        char    mappedStr[1000];  mappedStr[0] = '\0';
                        strcpy(mappedStr, mapEventParameters(rule, tLog, raStr));
                        char commandType[1000];  commandType[0] = '\0';
                        strcpy(commandType, getToken(mappedStr,0,TRUE));
                        if (((strcasecmp(commandType,"SELECT"))==0) ||
                                ((strcasecmp(commandType,"INSERT"))==0)) {
                                ExecutionModule::parseQuery( mappedStr);
                        } else if (strncasecmp(commandType,"CALL",4)) {
                                exeResult = app.executeCommand(mappedStr);
                        } else if (strcasecmp(commandType,"HOST")) {
                                HostObject          host;
                                exeResult = host.executeCommand(mappedStr);
                        }
                }
        }
        return exeResult;
} // executeCommand
```

Figure Execution Module - Multiple Action/Fail-Action clauses

If the clause is a DML query, it is handled in a similar manner to that of the condition clause, except that the DML allows a wider variety of operations i.e. insertion of objects are possible, as well as the SELECT clause, figure 6.17.

If an external action is required, this is signified by the CALL keyword, which then allows an application defined operation to be invoked. If the condition clause was not satisfied, then fail-action clauses (if present) are executed, in exactly the same manner as the action clauses.

Even though the REFLEX model had been designed as a parallel model, the prototype was still essentially sequential, the following section examines the feasibility of its concurrency.

6.5. Distribution and Parallelism

The architecture of REFLEX is such that when an event occurs a number of rules may be affected. As described earlier, the KMK instantiates the KSM to test if the event specification for a rule has been satisfied. The KSM cannot run concurrently, therefore the rules are tested sequentially. The speed and efficiency of the system would be improved dramatically if, in a multiprocessor or distributed environment, for every rule

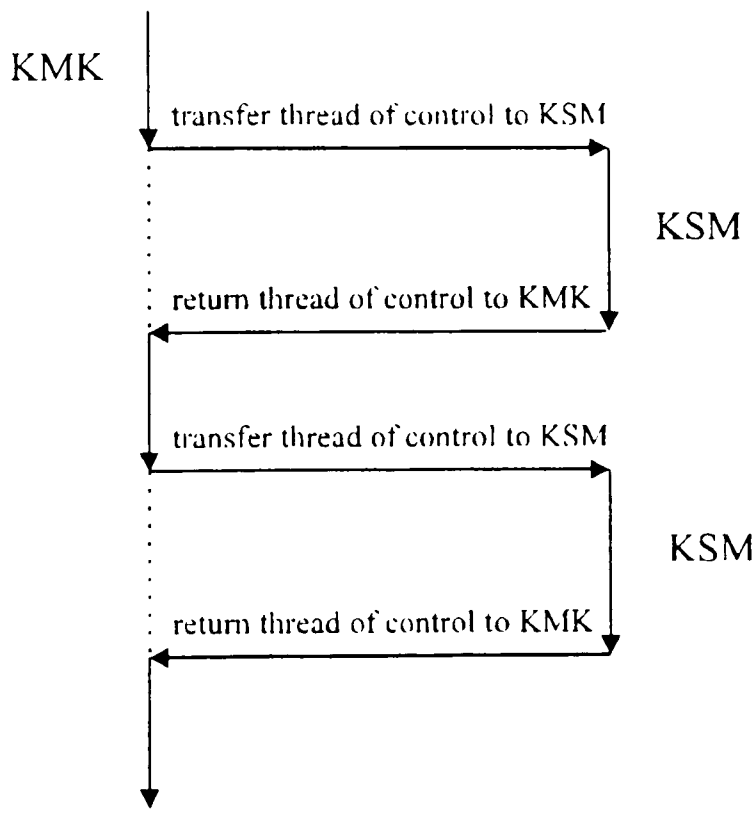


Figure 6.18 Existing Sequential Model

to be tested an instance of the KSM could be instantiated concurrently. This was the approach taken for a small concurrent prototype to test the feasibility of the models concurrency i.e. to parallelise REFLEX and specifically the KSM. The current local procedure mode of interaction between the KMK and the KSM, figure 6.18, can be described as follows: the KMK (or client) calls the KSM object (or server) and passes its arguments i.e. the event and affected rule. The KSM then takes control, carries out its processing, and eventually returns back control. At which point the results of the procedure are extracted and the caller continues execution. These steps are repeated for each affected rule.

The first stage at reducing the overhead caused by the KSM was to parallelise the module on the same host machine.

6.5.1. Possible Solutions

REFLEX was already being developed within a distributed unix environment, where processes run concurrently. Instantiating multiple processes is simple, since all that must be done are that a process is run in the background. Communicating between two processes is however, somewhat more difficult. There are essentially two approaches to interprocess communication, as described by Stevens [Stevens 92], using shared memory and message passing. With the trend to distributed systems, there has been a surge of interest in message-based interprocess communications. Unix provides a number of different forms:

- Pipes which are unidirectional paths over which processes may send streams of data to other processes
- Named pipes are permanent paths. Messages transfer discrete data elements
- Sockets are end points of two-way communication paths
- Remote procedure calls (RPC) allow a process on one system to call a procedure in a process on another system.

There are many further approaches that could have been taken to parallelise the module (e.g. parallel architecture database machines), but since REFLEX is a portable active database extension, most were ruled out as they could not be made portable. The adopted approach was to instantiate many KSM processes within the same machine using RPC since this method would be both upgradeable and portable.

6.5.2. Remote Procedure Call

The RPC approach is similar to the local procedure call, figure 6.18, in that one thread of control logically winds through two processes, one is the caller's process, the KMK, the other is a server process, the KSM, and waits for a reply message. The call message contains the procedure's parameters. The reply message contains the procedure's results and once received the results are extracted and the caller's execution is resumed. This approach does not solve the problem of single-tasking sequential execution of the KSM since the thread of control may only be active in one process at a time, therefore only one of the two processes is active at any given time. The RPC protocol does, however, allow for multi-threaded control where it is possible for the calling process to do useful work while waiting for a reply from the server. But this again is not a real solution since the only useful work the KMK would be doing is making more calls to the KSM. The KSM would then buffer these calls and execute them sequentially.

In order to achieve the desired parallelism, the REFLEX architecture implementation was *reengineerd to work in reverse*. Rather than making a call to the KSM server from the KMK client, the reverse scenario was required i.e. make the KSM the calling process and the KMK the server. This way multiple instances of the KSM, running concurrently, can call the KMK, figure 6.19.

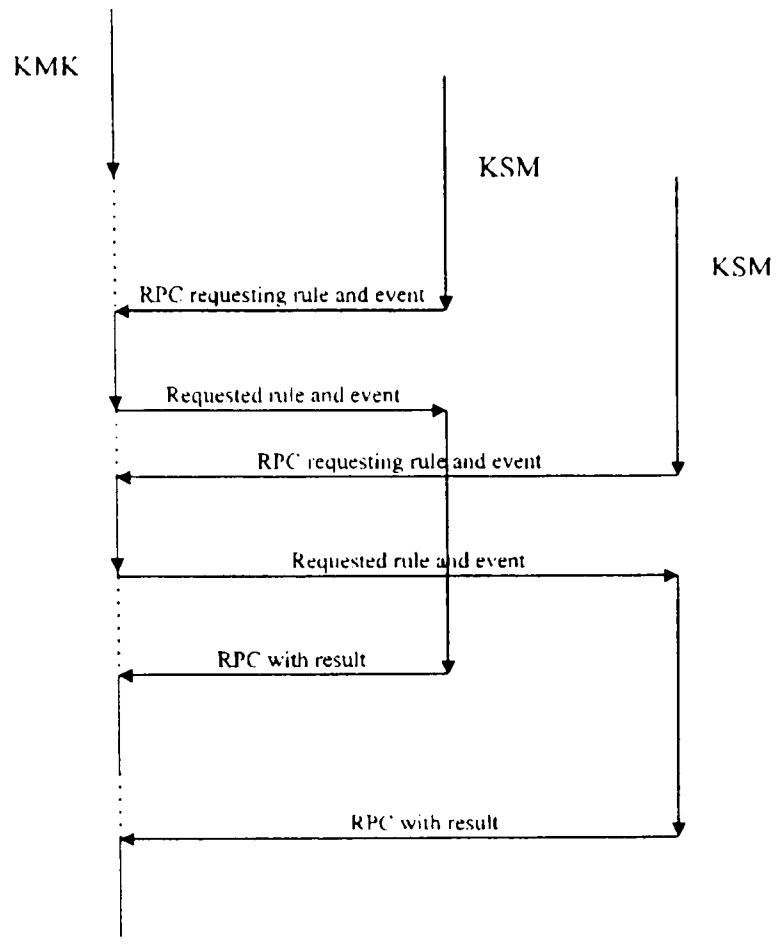


Figure 6.19 RPC Concurrency Model

The KMK instantiates the number of copies of the KSM that are required and then waits to service each instance of the KSM. Each KSM calls the KMK for the rule and event data that it is to test, which it tests and calls the KMK with the result. Once all of the KSMs have called the KMK with the results of their tests, the KMK stops acting as a server and moves on to its next task.

6.5.2.1. Implementation Details

For the concurrent prototype, the reimplementation involved a complete rewrite of the KSM so that it would initiate server requests and transfer data by means of RPCs. The KMK, unlike the KSM, is a large management object and hence only parts of it required changing (those that interacted with the KSM), i.e. new XDR (eXternal Data Representation, RPC implementation independent data types) data types were introduced along with handle information i.e. program number, version and procedure

numbers. The main method within the KMK which was responsible for scheduling the KSMs, *knowledgeScheduler()*, had to be completely rewritten. A KSM had to be instantiated for each rule, the name of the host and rule number that the KSM is to test are passed in the form of command-line arguments. The KMK would then continue its processing. When all of the KSMs have completed by calling the RETRESULT procedure, the knowledgeScheduler() loops through the results of the rules and acts upon those that were in-context. RETRESULT gets the process number and the result of the test, which it stores in the variable *result*, indexed by rule number. The procedure also increments the global variable *processCount*. It is this variable that the KMK tests to determine when to stop servicing the KSMs.

6.6. Performance

Widom [Widom 94] reports that current thoughts on active database usability revolve around the fact that the systems are just too slow for any real applications that require fast action, for example process control environments.

REFLEX has tried to address some of these areas by trying to build a fast reacting optimised system. But since benchmarking of the active database was considered to be out of scope at this stage, performance could not really be tested, but the designed features can be described as below.

Unlike other active database systems, for example HiPAC, ADAM, Starburst, POSTGRES, and Ode, REFLEX maintains indexes on all affected classes. This is possible because of its object-oriented architecture where all components are modeled as objects. These objects maintain links to other relevant objects i.e. rule objects maintain links to all events that affect it and objects that it rules upon, and vice versa. Hence when an event is raised, the system immediately knows whether it affects any rules. Only if it does, its execution is interrupted while the KSM determines whether

any rule has been brought into context by the event. If not, the current process is continued.

6.7. User Interface

The user or developer of an active application, using the REFLEX active extension, is presented with a fully object-oriented graphical user interface (GUI), the REFLEX Visual Supervisor (VIS). VIS aids the acquisition of domain knowledge by allowing the user to interact with it in a natural form.

Before introducing REFLEX's user interface, the interfaces provided by other active databases must be reviewed.

6.7.1. Related Work

Most active database user-interfaces are essentially text based. The user or more likely the application system developer enters the rules in the form of programming language code into a programming editor. The whole system must be compiled before the rules are available to the application. This implies a statically bound knowledge system, where the knowledge may only be entered prior to compile time.

Ode event-action (EA) rules, as described by Gehani, Jagadish and Shumeli [Gehani 92a] are entered under a '*trigger:*' section within an objects class header definition in C++. The triggers then have to be explicitly activated. This is accomplished by placing calls to the triggers, from the class constructor of an object. This means that the programmer must program the rules and compile the complete system for the triggers to be seen. Hence, not a task for the end user but for an application systems programmer. Ode does however, provide OdeView, a graphical interface to Ode. To

date, OdeView is for users who wish to use Ode without having to program extensively in O++. Agrawal, Gehani and Srinivasan [Agrawal 90], assert that OdeView does not cover the maintenance of Ode triggers, but the ability to create new and browse existing database classes. Starburst provides an SQL extension which allows for the declaration of rules. It has both trigger and condition clauses. It still has to be typed into programming language code and then compiled [Widom 91]. SQL statements are used for both the condition and action clauses. ADAM, as described by Diaz and Paton [Diaz 91b], provides a prolog interface to its active rule system. The user has to type in a prolog rule, which may be fired within the interpreted environment. The ADAM interface requires a user to be a skilled prolog programmer, generally outside the domain of the typical database user.

6.7.2. Vis Design Approach

Since the user is one of the most important components of an HCI activity, the approach adopted in REFLEX is that of ***user-centred design.*** The principles followed are those described by Lowgren [Lowgren 93] which state that the user's needs, rather than technical considerations, are the primary objectives. Not only must the system provide the functionality that meets the users' needs but it must be achieved in such a way that carefully considers ***how*** the user goes about a task.

Johnson [Johnson 92] reports that for the design of an effective HCI there are three main design stages (i.) knowledge acquisition/maintenance i.e. requirement analysis, (ii.) task analysis i.e. how the actual task is to be carried out e.g. entry of event, condition and action specifications, and (iii.) usability testing i.e. to the interfaces actually support the tasks. A number of authors present methodologies to perform the task analysis i.e. Annette et al. [Annett 71] propose the *hierarchical task analysis* method (HTA), which is essentially empirical where analyst study actual task performers at work. Diaper and Johnson [Diaper 89] propose *task analysis for*

knowledge description (TAKD) method, where task analysis was conducted at an abstract level which essentially allowed the acquisition of task knowledge. VIS was developed using the TAKD method, which allowed the system to be considered both at a high level i.e. as whole 'macro' techniques, and at low-level i.e. the requirements were further decomposed into smaller cognitive units (as a sequence of discrete tasks). Cognitive considerations e.g. reducing short-term memory load played a major part in the decisions taken in designing VIS. These considerations manifest themselves in VIS's feature of simple display and infrequent window motion amongst others i.e. the users will find that the windows appear in the same style throughout with certain functions and messages always occupying the same area of different windows.

The safety criteria for a system must be considered during its design process i.e. a balance must be found between automation and human control. Brown [Brown 88] asserts that depending on the relative capabilities of humans and machines, a task is either automated or left under human control. For example, in an air traffic control application, the human controller handles emergencies but other routine tasks e.g. progression along flight path etc. are left to the computer system.

Currently rules and events may be added and declared to the REFLEX system by means of VIS. The interface was prototyped and the process iterated until usability testing for task usability was successful. A brief introduction to the prototype visual interface is presented.

6.7.3. Visual Experience

The user is presented with the simple REFLEX Visual Supervisor menu, figure 6.20. If the user wishes to perform system administration functions i.e. add new rules or events, or to interrogate the system, they are all completed using this module. Access to the analysis functions are also available from the Analysis menu option.

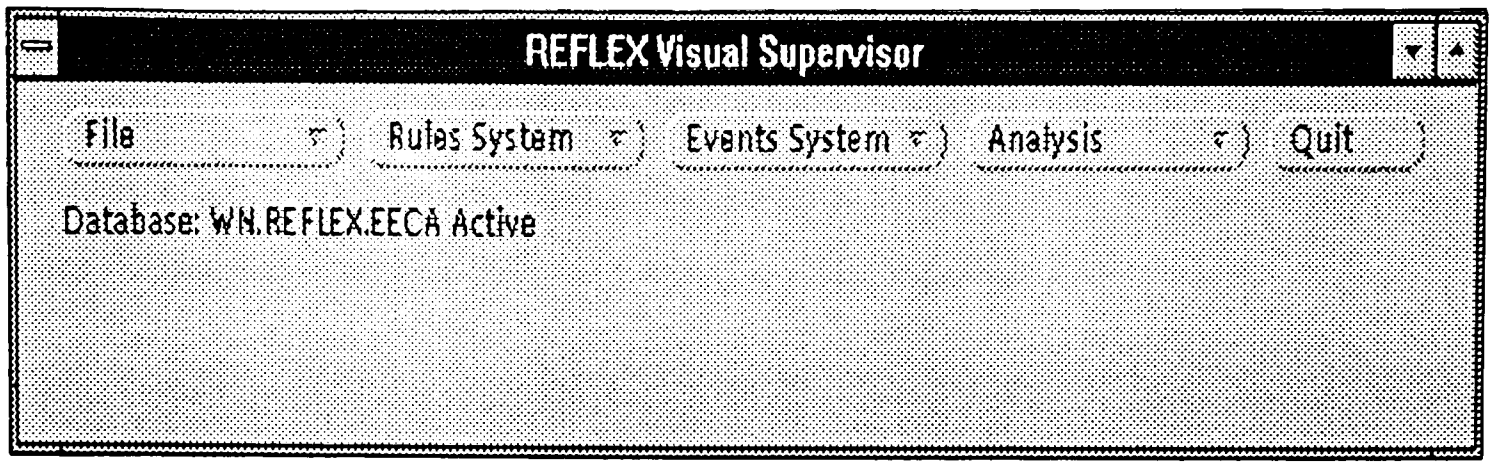


Figure 6.20 Vis Main Menu

When a user wishes to add a new rule, the New Rule option is selected from the Rule System menu. The user is presented with a New Rule Dialogue Window. The name and description of the new rule are entered. Following this, the English ESL clause must be specified. This is accomplished simply by entering the natural english statement. Once the English ESL has been entered, is parsed, and part-compiled, this causes the events that the affect the rule and the classes that the rule affects to appear in their respective windows, automatically. From the list of target classes, if the user selects a class, he/she is given the chance to select particular objects as targets or of exemptions from the rule's action.

The condition clause is completed using a full text editing sub-window. The system checks for syntactical and existence errors i.e. the referenced classes must exist, at this stage.

Since REFLEX uses the EECA knowledge model, there may be many action and/or fail-action clauses. To facilitate this, when the user enters the action list box, a new capture action window is opened. Into this window the user may enter the new action for the rule and its associated coupling and dependency modes. As described earlier, more extended query language statements are available for the action statement than are available for the condition clause. This is because the action clause may include additional query operations such as insert or delete.

Both the Condition and Action clauses can access the object that raised the event by

using the keyword OBJECT followed by its occurrence number in the event algebra expression. In figure 6.21 above, the action clause calls a user defined operator window called AlertOperator and passes the OBJECT1 as an argument, which in this case will be the actual aircraft object that has been updated.

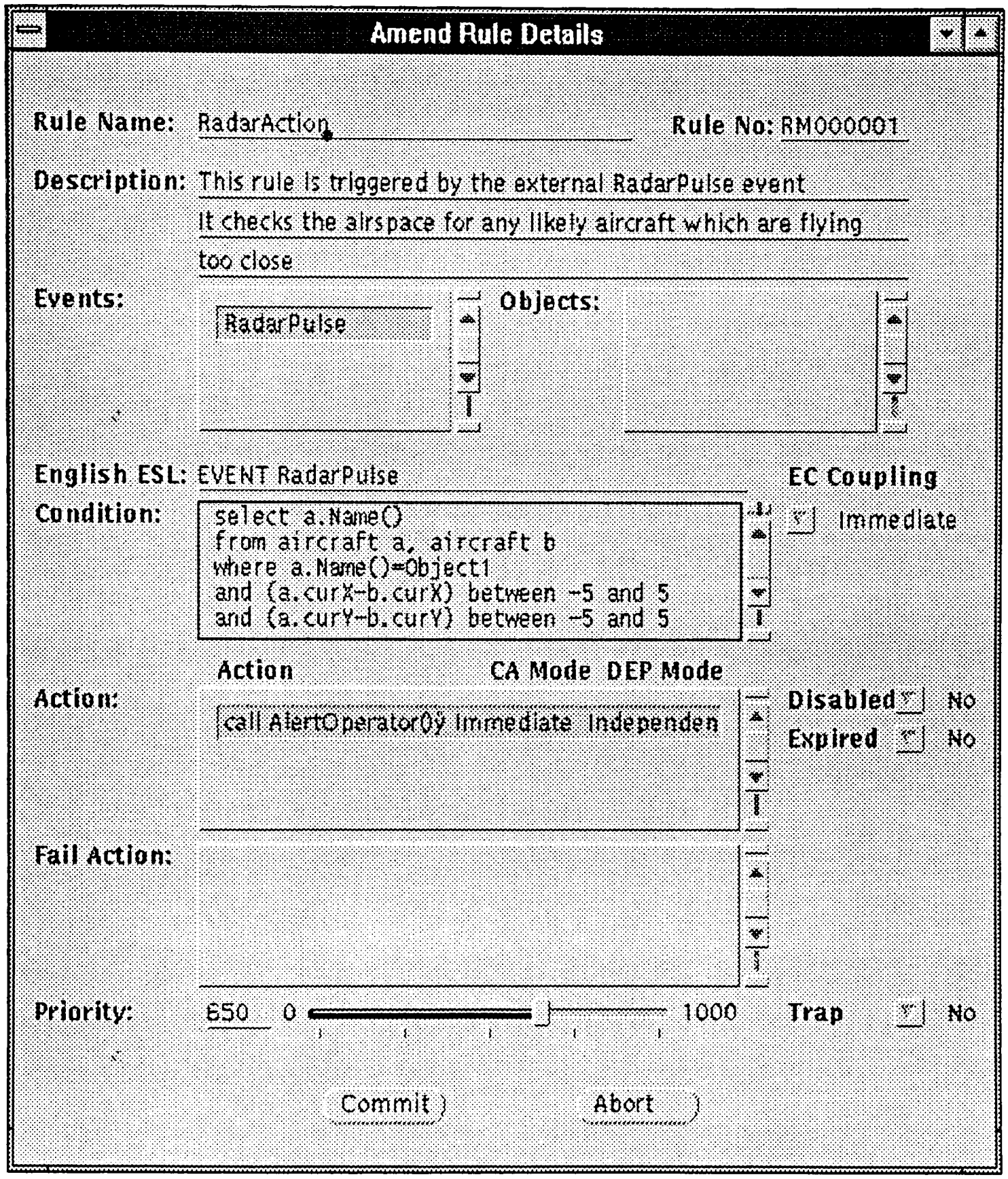


Figure 6.21 EECA Rule, Amend Rule Screen

The same intuitive approach is used for event management, i.e. declaration of new

user-defined events etc., figure 6.22.

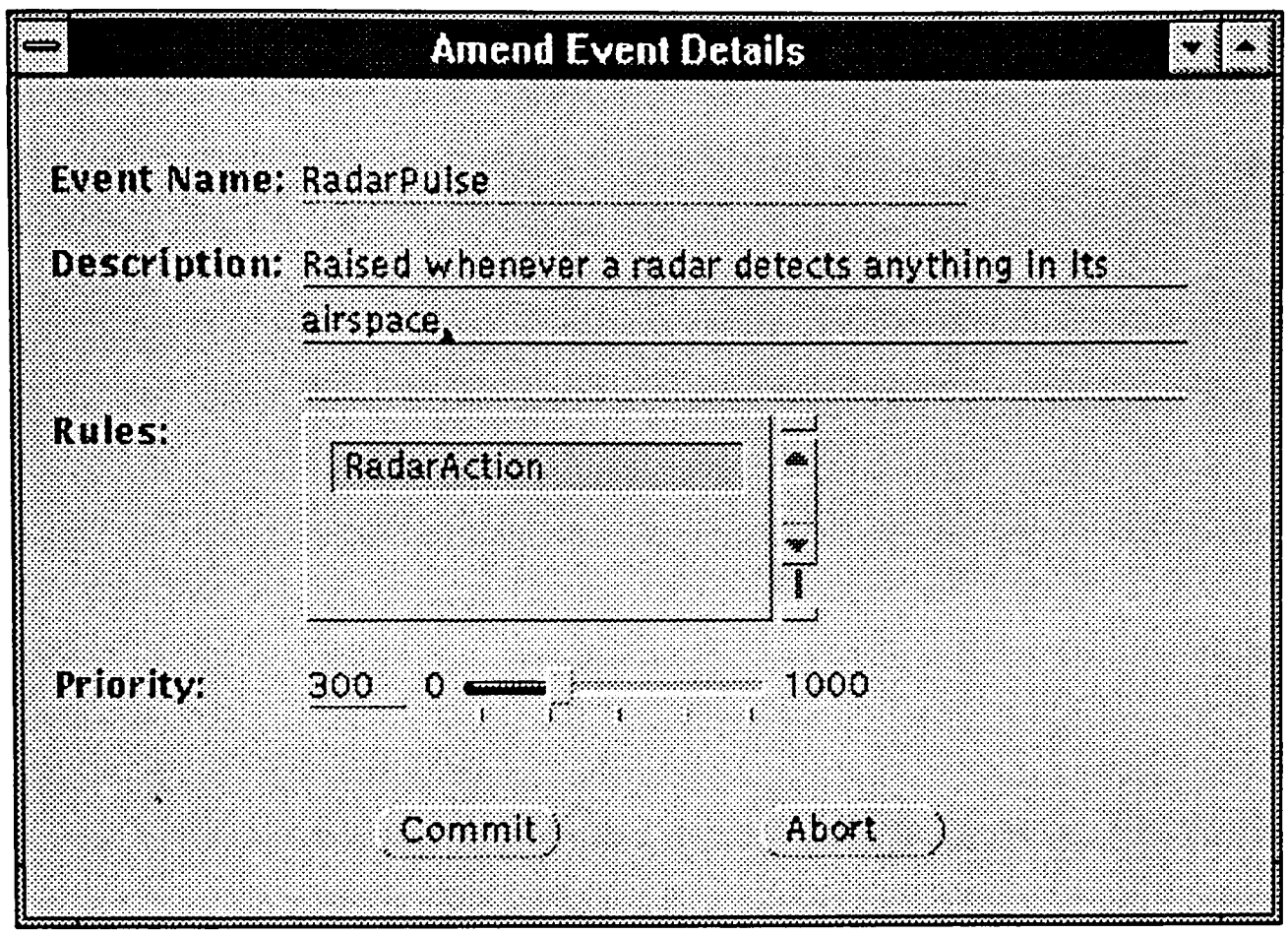


Figure 6.22 Dynamic Event Maintenance

A rule browser allows the user to investigate what rules are held within the system, their state, what objects they act upon and indeed, what events affect them.

6.8. Demonstrate Portability and Adaptability

A major goal of this research was to provide a portable and adaptive active database layer for a given host object-oriented DBMS (ODBMS). This being the case, Kim [Kim 95] reports that there is no standard definition for an ODBMS even though the Object Management Group[3] is trying to forge a standard object model. For this research, a number of standard features are assumed present in all ODBMSs, these being objects, their identity, inheritance, aggregation, object and class information,

[3]An industry consortium that is seeking to define a standard object model, and standards for interaction.

collections, and navigational query facility.

In order to realize this goal of portability a generic framework was required. This was formed for the initial development and used for the second implementation, which allowed a structured approach to the porting process to be taken. It can be described as follows:

After successful implementation of a prototype on the first host database, ONTOS

1. Porting

   The goal of this stage is to make the component modules compilable on the new platform.

2. Adapting

   Make each component module work by adapting REFLEX to the host database, so that the active functionality works transparently.

3. Extra Functionality

   If the host database does not provide certain required functionality, add this.

4. Component Integration

   Make the modules work together.

5. Testing

   Test the implementation by repeating the above steps, but for an application system, and then execute the application

The first goal of portability was, theoretically, relatively simple since modern day open systems/software engineering practices dictate that software development should be

conducted in such a way so as to produce platform independent source code. This simply translates to using freely available development tools and languages and taking account that the system may reside on a different platform in the future, i.e. ensuring that the platform dependent code in a layer that can be changed. For this research the programming was conducted in C++ [Stroustrap 86] for which compilers are freely available on many (if not all) platforms.

The second goal of adaptability is not as straightforward. Since the active layer must function in the native mode of the host DBMS i.e. it must be transparently active and adapt to the host.

These two goals are realized in the second implementation which is described in the following section.

6.8.1. The Porting Process

The implementation framework described earlier was adhered to in order manage the entire porting process to the second platform, POET.

The first stage was the raw porting of the program code so that it compiles on the new platform. REFLEX being implemented in C++, and both the ONTOS and POET DBMSs being essentially C++ class libraries, the porting process between the two should have been, theoretically, straightforward. There were problems however as the ONTOS version used AT&T C++ 2.1 precompiler which has its many quirks, and the POET version used MicroSoft Visual C++ 1.5 (referred to as VC++). The VC++ claimed it was draft ANSI compliant but this proved inaccurate. The compiler differences were however minor problems which were easily resolved.

POET attempts to hide the additions it makes to the C++ programming environment,

by using its PTXX pre-compiler, to add the extra database code to the application code before it is compiled. POET implements a schema database into which all persistent classes must be classified, since C++ does not support run-time type information, this is generated and maintained by PTXX.

PTXX, was however, fairly bug-ridden. For instance the precompiler did not like the C++ keyword *const*, all references to const had to be changed back to the old C programming language [Kernighan 78] style of *#define*.

The PTXX pre-compiler is the mainstay of the POET product. It reads in C++ class definition header files, which must have *.hcd extensions, from which it generates the *.hxx header files and the class factory files (base.hxx and base.cxx) which are used to build database objects, figure 6.23. A further file produced is the poet2.hxx file which is the C++ representation of the database objects. The use of the different types of files introduces problems and will be described later.

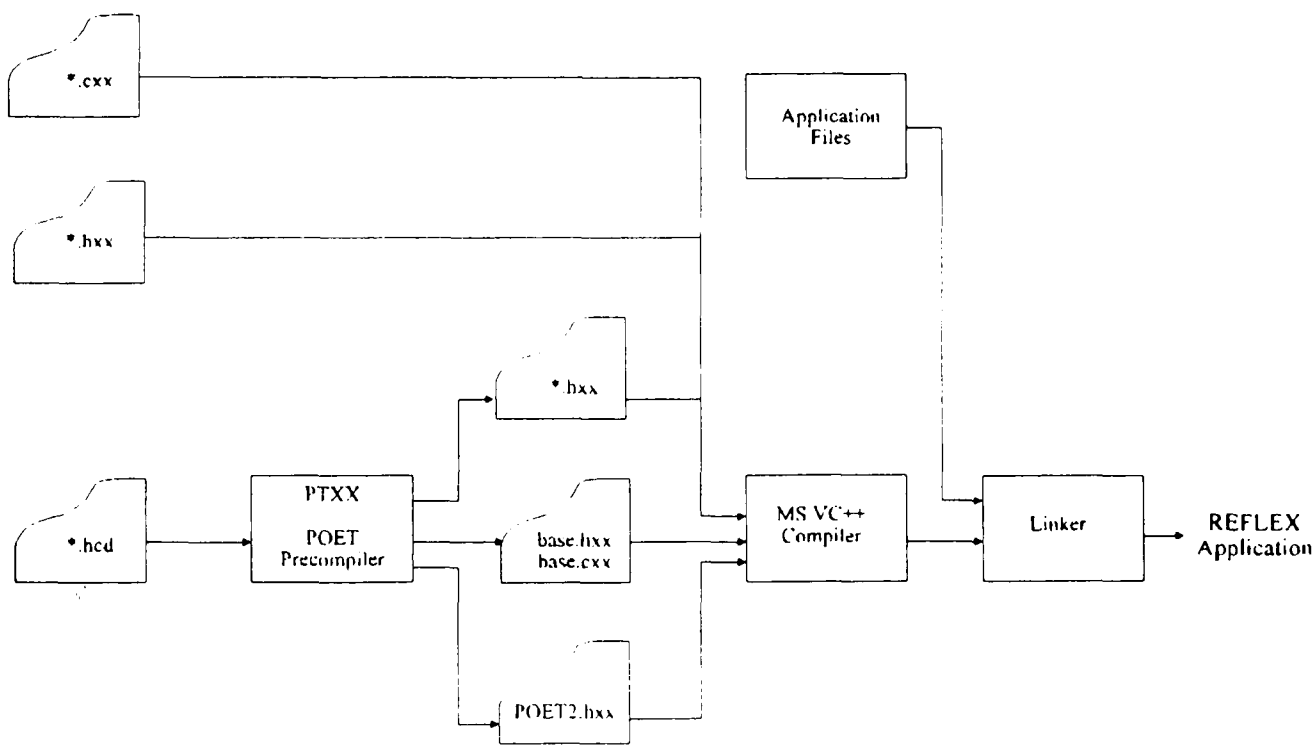


Figure 6.23 POET Compiling Process

Similar to ONTOS, POET classes must inherit from a persistent base class, in this case *PtObject*. For a class to become persistent in POET, it must be declared with the *persistent* keyword. This enables the precompiler to know when to add the inheritance form PtObject to the class declaration.

In order to modify the program code so that it would at least compile, the following changes were required:

- the keyword *persistent* was added before each class that was to be persistent and also to forward references to the class. This keyword was picked up by the PTXX pre-compiler which changed the program code to show that the persistent class was actually a sub-class of PtObject. For example:

  persistent class person

  was replaced by the PTXX to:

  /* persistent */ class person : public PtObject

  But if it was actually declared as being inherited from PtObject, the pre-compiler would report an error.

- C++ constants that were declared using the *const* keyword, were not supported by the PTXX pre-compiler. Instead these had to be reverted to the form used in C where constants were declared using pre-processor directives i.e. they were defined in the pre-processor only using the *#define* statement.

- within a class, character strings are declared as pointers to a type char i.e. *char**. In POET for these sub-parts of a class to persist with the rest

of their class they have to be declared as *PtString*. All of the references to char* had to be changed to type PtString.

- A very major problem was that POET required persistent classes to be declared within headers with an *.hcd extension. The standard header declarations were still to be held in *.hxx files. The pre-compiler would produce further output files base.hxx and base.cxx, which would both reference the .hcd and the .hxx. The body of the classes would be stored in standard *.cxx files, which would reference all of the above mentioned file type. This, because of all of the different files and their extensions, caused many problems of circular references i.e. where one file would reference another file which referenced the first file.

6.8.2. The Adaption Process

After the program code compiled under POET and MSVC++ the next step was to adapt the code so that the REFLEX extension was a transparent addition, i.e. POET users should not have to change the way in which they interact with the host database.

In order to adapt REFLEX to work with POET, there are a number of stages:

- Change the database access mechanism

  for inheritance from the base class

  e.g. ONTOS class newClass :public OC_Object

  POET *persistent* class newClass

  for the event signal generators for internal database and transactions events

  e.g. ONTOS has putObject()

  POET has Assign() followed by Store()

- Add active base class

  To convert a POET persistent class to an active class, the active class must be explicitly declared

  e.g. change persistent class person

  to persistent class person : public AObject

  This is because the POET precompiler always assumes that the persistent class is derived from PtObject, its base class.

- Saving the schema definition, to allow run-time type checking

  ONTOS provides a powerful classify utility, which takes standard header files and saves the definitions, in the database.

  POET requires specially named header files, which after being processed by the PTXX utility, generate a number of files, and may only be read in by certain programs. Hence, many programs require processor directives to select which type of POET file to include.

The adaption was generally localised to the database access functions, since these would be used by the database programmers.

6.8.3. Extra Functionality

It was discovered that POET did not provide a type manager. This would allow the user to ascertain as to the exact type or class a given object belongs to at run-time. This information is not required by the average application of an object database because the application already knows what type of object it retrieves into memory. But for an active database application this is very important because, internal events are caused by some interaction within the database e.g. an object may be read or is being written to the database. The event detector must know exactly what type of object raised the

event and whether it appears in the event specification clause of a rule.

This deficiency required a type manager to be written specifically for POET, in order to access user types. This was achieved by reverse engineering the output database that POET produces, in order to specify functionality for a type manager.

ONTOS provides a navigational query facility as well as an Object SQL dialect. POET also provides a navigational query language but does not provide any SQL functionality. This is required, since REFLEX provides a high level generic SQL interface. A small Object-SQL interpreter was constructed to allow SQL queries. For simplicity, the application class and attribute names were 'hard-coded' into the SQL program.

6.8.4. Component Integration

After all of the modules were ported and adapted to the new platform individually, they were integrated together, one by one, using dummy interfaces for any other required modules, so that the system could be tested whilst being constructed.

6.8.5. Testing

REFLEX is a prototype system, and is continually changing as new ideas are tested. As such real consumer testing was not undertaken. The model was however put through an inbuilt test harness, on every major change, it would create 10,000 rules systematically and a number of objects to test against simply to ensure that the system is reasonably reliable.

6.8.6. What was learned in the Porting Process

Before the porting process had begun, there was an assumption that all databases of a particular genre would have a certain set of base functionality. This assumption was proved to be unfounded. The two sample databases had some differences, which are tabulated in table 6.1.

Features	**ONTOS**	**POET**
Type Manager	Yes	No
Navigational Query Facility	Yes	Yes
Object SQL	Yes, Limited	No
Direct Links	Yes	Yes
Transparent Links	Yes	No
Explicit Requests only	Yes	Yes
Container Classes Array Lists Dictionaries Bags	 Yes Yes Yes Yes	 No No No Yes
Transaction Classes	No	Yes
Persistent Strings	Normal	Special Class

Table 6.1 Object database feature list

The process of porting and adapting REFLEX to POET was straightforward. Its core functionality ported in four days. After it was discovered that required functionality i.e. the type manager, was not present in POET, this had to be implemented in order for the system to work. This required the reverse engineering of POET database files to identify the structure of the object representations, and took three days.

Overall the implementation on ONTOS was much cleaner and simpler than the porting

onto POET. This was because ONTOS (i.) adhered to the standards i.e. ANSI C++, (ii.) it was much more robust, and (iii.) not as long-winded as POET, for example, each change to a header file in POET, meant a complete re-compilation of the schema.

6.9. Summary

This chapter described the design and implementation of the REFLEX model. The REFLEX model is a portable and adaptive extension to an existing host database. The chapter introduced the two host DBMS for the prototype implementations: ONTOS and POET, highlighting their differences.

REFLEXs architecture was introduced briefly followed by a dissection of its components, and their interactions, supported by code segments.

The models novel concurrency methods, and distribution were described. These were then followed by a sample concurrency prototype using RPCs, which illustrated that the model had to be changed to use these effectively.

The user interface, VIS, was introduced together with some of its intelligence features i.e. detecting which events would affect a rule, and which classes a rule affects.

The process of the second implementation highlighted some interesting observations, such as what is an object model? This arose because two object databases were used in the respective prototypes, but each had slightly different characteristics.

Chapter 7

Evolution and Experience of REFLEX

This chapter illustrates the various aspects of the REFLEX model and the development of its prototypes by means of example applications.

7.1. Introduction

The REFLEX model has been described in the previous chapters. This chapter examines the various working prototypes that were constructed in order to realise and provide feedback into the resultant system. It goes on to illustrate how the various features of the resultant model and prototype may be used, for example how the rules and events are declared, the range of events that may be specified for a rule's event clause, how the condition and action clauses are specified.

The chapter is structured as follows: section 7.2 reviews the prototypes produced in the development of the final system. Section 7.3 introduces the use of the rules' system within REFLEX, followed by section 7.4 which illustrates these features by way of two example applications. Section 7.5 examines how the functionality of the prototype meets those proposed in the REFLEX model. Finally section 7.6 summarises the chapter.

7.2. The REFLEX Prototypes

The REFLEX model was formed by the implementation of a series of working prototypes. The lessons learned from each prototype were then addressed and the functionality incorporated into the subsequent prototype. The evolution of the prototypes can be seen in table 7.1.

Prototype No	Description
1	Rules represented as objects and simple events represented as system attributes
2	Events represented as first class objects
3	Implementation of the Vis graphical user interface
4	Complex Event Specification Language and parser
5	Events maintain references to Rules
6	Implementation of Complex Events
7	Self-Activity
8	Non-Destructive Knowledge
9	EECA Knowledge Model
10	Concurrent Distributed Model

Table 7.1 History of Prototypes

The objective of the first prototype was to provide a minimal active capability to an existing commercial object database. It adhered to the ECA knowledge model, and represented rules as objects but events as both application and active database system

attributes, similar to the approach taken by HiPAC. Once constructed this approach proved unsuitable because the maintenance of the events was problematic since each event had to be declared at compile-time, which was an inflexible solution.

This desire for flexibility and uniformity fed into the design of prototype two, where events were modelled as first class objects in their own right. This meant that events could now be declared dynamically at runtime as opposed to statically at compile-time, and that the same underlying maintenance structure could be used for rules and events as well as data (see discussion in sections 5.5.1, 5.5.2).

Interaction with the prototypes was essentially through a programmatic interface, even though all processing was handled internally to the rules. This meant that testing the prototypes or knowledgebase was a time-consuming process since each change involved editing, recompiling and the re-linking of the system. A graphical 'point and click' user interface that was independent of any application was deemed necessary, to allow realistic prototype testing. This was the rationale for prototype three. The Vis user interface was developed which would operate independently alongside any user application, and allow the application designers to enter domain knowledge in the form of rules dynamically. This feature, to the best of my knowledge, is not provided by any other active database prototype.

So far, the prototypes only handled simple events, the resultant system should be able to handle complex events. Prototype four was constructed which had the goal of implementing the English ESL (Event Specification Language), in terms of a parser and syntax/semantic checker. The prototype allowed the semantics of the ESL to be tested by trying different permutations of specifications, which verified that the semantics of the ESL are well defined and precise. The prototype did not however attempt to implement complex event processing, just the preliminary declaration and parsing of the complex event specification.

All of the implemented prototypes highlighted a major performance issue. This can be explained as follows: when an event is raised it may affect many rules, but which ones? Each time an event was raised all of the rules had to be tested to determine if the event had indeed affected the rule. This process was inefficient. To overcome this inefficiency, prototype five was constructed which allowed each event to maintain a reference to each of the rules it affected and vice-versa. This prototype was indeed operationally faster, since each event knew which rules it affected, if any, and this knowledge became more significant as the size of the rule base grew. This approach was novel and was not reported in the literature, to the best of my knowledge, since the majority of related research still modelled events as system attributes.

Even though the syntax and semantic parser for specifying complex events was implemented in prototype four. The actual complex event handling, detecting and evaluation was implemented in prototype six. This involved building a complex back-end system, which would test rule ESL statements for satisfaction, according to the language constructs available in ESL, including logical, temporal and validation statements.

Prototype seven not only provides the two main goals of active databases, i.e. to minimise code redundancy and to respond within time to any situation, but it also utilises these goals to maintain itself, i.e. its application knowledgebase. This was achieved by allowing internal system management to be encoded in the form of rules, i.e. to become self-active. This proved a novel and successful strategy, since on the creation or amendment of a rule, the system would become active and check that the constituent parts of the rule were declared correctly, i.e. both syntactically and semantically checked, and then part-compile the rule.

Whilst building example applications, creating rules on objects, testing and firing the rules, then amending the rules, e.g. the condition test for a rule that monitors aircraft movements may have highlighted aircraft that were separated by 5 miles, later this was

changed to test for aircraft separated by 10 miles, it became apparent that knowledge was being lost. A rule fired against the object at different times could have completely different semantics. For example, if a patient has a headache, the doctor may prescribe salicylic acid (Anadin). Later, when the side-effects of Anadin on children became known, the same doctor may, for the same symptoms of headaches, prescribe paracetamol (Panadol). This concept, of the revision of knowledge, led to the creation of prototype eight, which incorporated the concept of non-destructive knowledge (see discussion in section 4.7.1), where once a rule had been fired it was not amendable. If an amendment is required, a new rule must be declared which is pointed to by the old rule.

Whilst building applications, such as the air traffic control system, it was noted that sometimes it would be convenient to test an external application condition, e.g. the current angle of a radar. With conventional active database systems, in order to access this information dummy updates are required on the database so that the state may be tested since only internal database states can be tested. The testing of external states is required. Also, it was noted that if a number of actions are required on the occurrence of similar situations, a number of rules need to be declared, which also implies a significant amount of redundant knowledge. There is an overhead for each of the rules which have the same situation to be tested. This lead to the creation of the EECA knowledge model, which allowed for an extended scope for the condition clause, i.e. external condition testing, and multiple actions for a given situation, but each with their own coupling mode and dependency. This EECA was embodied in prototype nine. After building example applications with the EECA knowledge model, it was also recognised that sometimes if the condition for a rule failed it would be useful to allow alternate actions. These multiple fail-actions were then embodied into the EECA knowledge model and prototype.

Even though REFLEX was designed as a distributed parallel model, this was not realised in the prototypes. Prototype ten was constructed to test the adopted distribution

approach, that of using remote procedure calls to instigate processes on the same and different host machines. This prototype demonstrated that the design of the system had to be modified since the design assumed that the central kernel (KMK) would be the client and the called modules the servers. Instead, the called modules had to be instantiated by the KMK, and then they became clients of the KMK server, i.e. the prototype functionality proved to be the opposite of the original design.

The process of building the various prototypes provided valuable insight, and feedback into the system. It also uncovered a number of issues such as, should events be modelled as first class objects, what form the ESL would take, the requirement for reference to rules to be maintained by the events, self-activity, the concept of non-destructive knowledge, the attributes of the EECA knowledge model, and the method of distribution.

The following section examines how the final prototype should be used.

7.3. Using the Rules System

Rules within REFLEX are declared dynamically as and when required. The application designer is at liberty to utilise either a bespoke programmatic interface, or the VIS graphical interface when specifying the rules. This choice makes no difference to the rules system because the processing is conducted within a rule object and not at the interface level. This can be exemplified by reference to the program code, figure 7.1, which illustrates how a REFLEX user could capture and then declare new rules, dynamically, by sending relevant messages to the rule object which would then test the arguments for syntactic or semantic errors. The example program code can be described as follows: after the variable declarations, the first statement checks to see if the rule already exists. If it does the routine exits. Following this is user interaction code to capture the constituent rule details, the subject of the next section. After the details have been captured, the rule object is informed by sending messages to it, e.g.

rule->parseEventSpec (ESL, errorPos, errorBuffer);

Here the rule is sent the message parseEventSpec which will take an ESL string as an argument and parse it for both syntax and semantic correctness.

The final message to the rule is to store itself in the database, and is simply:

rule->putObject();

```
void add_rule() {
        char    name[100];
        char    ESL[100], condStr[200], exeStr[200];
        char    *desc[3], description[3][100];
        char    ch;
        Rule    *rule;                                  // Rule Object ------
        int     evSpecPass = FALSE;

        // Textual rule details capture interface
        printf("\nPlease enter the new rules name\n"); scanf("%s",name);

        rule = (Rule *) OC_lookup((char*)name);         // Does it exist already
        if (rule != NULL) {                             // Yes, then exit
                printf("\n\nRule : %s Already exists on the database! \n\nAborting!\n\n", rule->Name());
                return;
        }

        printf("\nPlease enter description line 1: ");
        get_input(description[0]); desc[0] = description[0];
        printf("\nPlease enter description line 2: ");
        get_input(description[1]); desc[1] = description[1];
        printf("\nPlease enter description line 3: ");
        get_input(description[2]); desc[2] = description[2];

        printf("\nPlease enter Event Specification: ");       get_input(ESL);
                // e.g. update aircraft between 16/3/95-18/3/96

        printf("\nPlease enter Condition String, please ensure to put ';' to finish ");
        get_input(condStr);

        printf("\nPlease enter Action String, either as a SQL query of a function call\ni.e. select a.ID() from
aircraft a where a.Name() = OBJECT1;\nplease ensure to put ';' to finish\n or\tAlertOperator");
        get_input(exeStr);

        // Actual rule declaration ------------------------------
        REF_transactionStart();
                ADBGetRM();
                rule = new Rule((char*)name, RM);       // instantiate new rule
                rule->Description((char *[]) desc);     // declare description
                // declare ESL clause
                char errorBuffer[200]; errorBuffer[0] = '\0'; int errorPos = 0;
                evSpecPass = rule->parseEventSpec(ESL,errorPos, errorBuffer);
                if (!rule->conditionStr(condStr))       // declare condition
                        printf("\ncondition string FAILED....");
                rule->actionClause(exeStr, 0,0);        // declare action 1
                rule->actionClause("AlertOperator", 0,0); // action 2
                ...                                     // other attributes i.e. CA coupling mode
                rule->putObject();                      // store rule
                printf("\nCommitting Rule details\n\n");
        REF_transactionCommit();
} // add_rule
```

Figure 7.1 Program fragment to capture rule details

7.3.1. Constituent Parts of a Rule

A rule is made up of a number of constituent parts, i.e. the name and description of the rule, its triggering event specification, its condition clause, any number of action statements, its event-condition coupling mode, and its priority, each of which need to be captured.

7.3.1.1. Declaration of Complex Events

The triggering complex event for a rule is specified, as described in chapter five, in the following form:

> [NOT] Event1 [AND|PRECEDES|SUCCEEDS|OR] Event2 [WITHIN T SECONDS|ON DATE dd/mm/yy|AT TIME HH:MM]

Once this ESL expression is entered and is passed to the rule, the rule parses the expression for both syntax and semantic correctness, i.e. not only must the syntax be correct but any references to any other objects must exist.

7.3.1.2. Specification of Rule Condition

The condition is entered as a string which may be a call to an external condition module e.g.

```
call getTemperature
```

or an OSQL statement e.g.

```
select temperature
from vessel
where vessel.id() = Object1;
```

7.3.1.3. Event-Condition (EC) Coupling Mode

The EC coupling mode, is entered as an argument to the *rule->ECcoupling* message, the arguments being *immediate*, *deferred*, and *decoupled*, e.g. in the air traffic control scenario if the radar causes a pulse event then the condition should be tested immediately i.e. rule->ECcoupling(immediate).

7.3.1.4. Action Clause Specification

The action clause is captured in a string similar to the condition clause, except that when it is passed to the rule both the action's CA coupling and dependancy modes are also passed. A final optional parameter, a flag to indicate whether it is an action or a fail-action, may also be passed as arguments of the message.

```
void add_event()
{
        char                    name[100];
        char                    *desc[3], description[3][100];
        char                    ch;
        EventObject             *new_event;

        printf("\nPlease enter the new aircrafts name\n");
        scanf("%s",name);

        new_event = (EventObject*) OC_lookup((char*)name);

        if (new_event != NULL) {
                printf("\n\nEvent : %s Already exits on the database! \n\nAborting! \n\n",
new_event->Name());
                return;
        }

        printf("\nPlease enter description line 1: ");
        scanf("%s", description[0]); desc[0] = description[0];
        printf("\nPlease enter description line 2: ");
        scanf("%s", description[1]); desc[1] = description[1];
        printf("\nPlease enter description line 3: ");
        scanf("%s", description[2]); desc[2] = description[2];

        OC_transactionStart();
        new_event = new EventObject((char*)name);
        new_event->Description(desc);

        ADBGetRM();
        new_event->putObject();
        OC_transactionCommit();
}       // add_event
```

Figure 7.2 Program fragment to capture event details

7.3.2. Creation and Declaration of Events

The application system programmer or user may create new events at will, dynamically. These are set up similar to the creation of rules, i.e. the program code would capture the event description details and then send messages to a new event object, see example program code to capture event details, figure 7.2.

```
#include <refrouts.h>
#include <userrout.h>

// ADD APPLICATION INCLUDE FILES BELOW
#include "atc.h"

// Add Additional Commands to enum list
enum Commands  {        NoMatch = 0,
                        SELECT          = 1,
                        AlertOperator = 2
                };

// REFLEX required Routines ----------------------
char* AppObject::extractCommand(char* str, char* restOfArgs) {...};

// Add application member names below
int AppObject::syntaxCheck(char * commandStr) {
        char    args[1000], commandArray[200];
        char*   cmdStr = &commandArray[0];

        strcpy(cmdStr,extractCommand(commandStr,args));
        if (strcmp(cmdStr,"select") == 0)               return SELECT;
        if (strcmp(cmdStr,"AlertOperator") == 0)        return AlertOperator;

        // Default NoMatch
        return NoMatch;
} // syntaxCheck

// Add function names below to command dispatcher
int     AppObject::executeCommand(char * commandStr) {
        int     cmd = syntaxCheck(commandStr);

        if (cmd == NoMatch || cmd == SELECT) return cmd;

        char    args[1000];
        char*   cmdStr = extractCommand(commandStr,args);

        switch (cmd) {      // Command dispatcher
                case AlertOperator: {
                                ATC atc;
                                atc.AlertOperator(args);
                                break;
                                }
                default: break;
        } // switch

} // executeCommand
```

Figure 7.3 Program fragment to define external actions and conditions

In order to raise the event, the application program code will require explicit calls to the event object.

7.3.3. Definition of External Conditions and Actions

REFLEX allows for both external conditions and actions. In order to utilise this facility, in the current prototype, the user adds references to the external program routines into a *system hook* module. This module (called userrout.c), may if desired incorporate the complete external logic for a routine, but generally will call the routine from other user defined modules. It, and other user defined modules are then compiled and linked into the final executable application program.

The REFLEX system code is not recompiled, it is distributed as system binaries (machine executable code).

e.g. a portion of the program module userrout.c for the ATC application segment showing AlertOperator(), figure 7.3.

7.4. Example Applications

Two diverse application domains are considered within this section (i.) a real-time scenario, that of Air Traffic Control Systems, and (ii.) an administrative system, that of a university's Student Records System. Both of the domains are first introduced in terms of current use with traditional non-active databases, and how they may require distributed logic.

7.4.1. Air Traffic Control System

An Air Traffic Control System (ATCS) consists of a number of sub-systems: the controller, the radar, the aircraft, its environment, and the ATCS itself. These sub-systems exhibit independent behaviour and concurrently interact with each other.

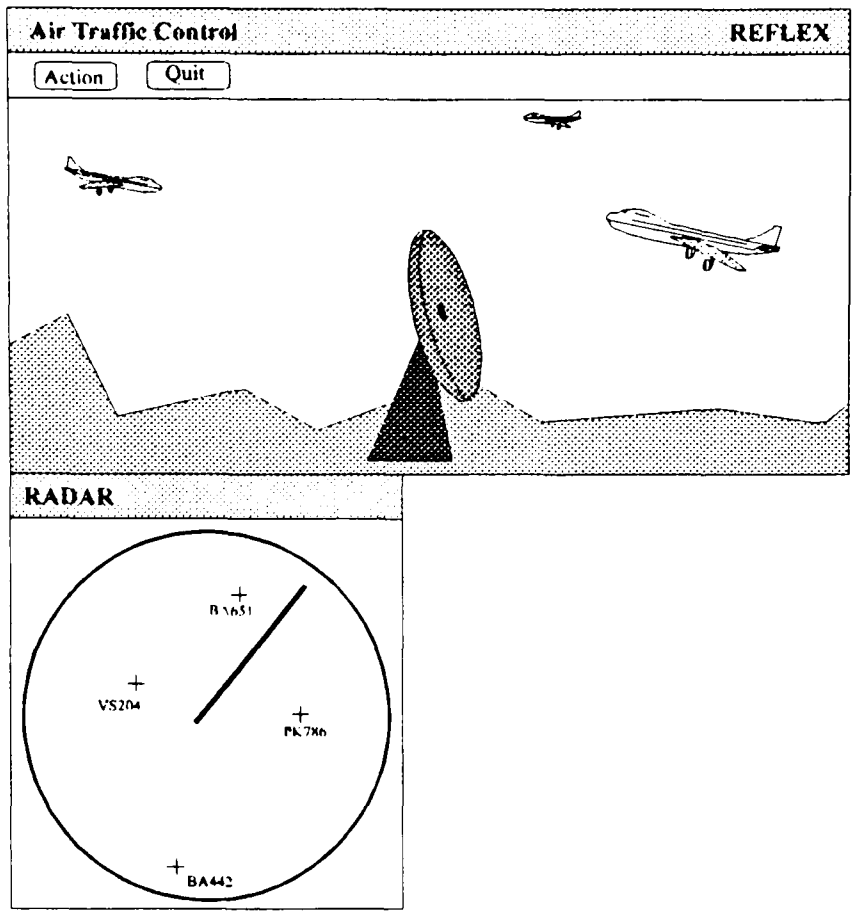


Figure 7.4 Air Traffic Control Simulation

7.4.1.1. Traditional Approach

In an air traffic control system, the operators need to know when an aircraft enters their airspace. When an operator manning a radar, observes an aircraft entering the airspace, the operator makes contact with the aircraft, and requests certain details from the pilot. These details are noted down on cards. The details or attributes for the aircraft would be, for example: the aircraft registration number, the flight number, the exact location latitude and longitude, its altitude, its destination, its bearing, its ETA. Similarly for aircrafts that are taking-off or landing within the airspace, but out of scope of the radar; the aircrafts details need to be recorded. The operators would then have to track the aircrafts, making sure that the aircrafts adhered to their prescribed flight-paths.

This task can be automated, whereby the radar, feeds signals direct to the database of current airspace. Application programs would then update information, and other monitoring programs would then poll the database to ensure that the aircrafts are on their prescribed flight-paths. Application programs would also periodically *poll* the database to ensure that aircrafts do not get too close to one-another.

7.4.1.2. Active Approach

Using REFLEX, when a plane enters the airspace, the radar will produce a pulse. REFLEX, on receipt of the radar signal (external event), will open a dialogue screen with the operator, requesting that the operator make voice contact with the aircraft. Before this stage, the ATCS active application will try to read the aircrafts transponder signal. If successful, will inform the operator via the dialogue screen. Once the aircraft's details have been entered, the object class of aircraft, may have rules attached. Such as:

```
ON      <radar.aircraft_moved>
IF      aircraft.location does not approx.
        equal prescribed flight-plan
THEN    CALL alert operator

ON      <radar.aircraft_moved>
IF      aircraft in vicinity of another aircraft
THEN    CALL  alert operator
```

Thus, the database itself would keep the aircraft's position updated in real-time, on radar events such as *aircraft position changed*. The database would monitor the aircrafts position and those of any other aircraft in the vicinity. Informing the operator of the current status in the airspace and alerting the operator if an unforeseen or hazardous situation arises.

A rule to test whether an aircraft which has changed its position is in danger, by moving too close to another aircraft, could be brought into context after an update to

the database by a simple/primitive event. An OSQL query tries to determine whether the aircraft in question is in the vicinity of another aircraft. If so, the operator is alerted, and a log entry made. An example rule for this scenario could be:

```
E     AFTER UPDATE aircraft
C     SELECT        a.Name()
      FROM          aircraft a, aircraft b
      WHERE         a.Name() = OBJECT1
            AND     (a.CurX - b.CurX) BETWEEN -5 AND 5
            AND     (a.CurY- b.CurY) BETWEEN -5 AND 5
            AND     (a.CurZ - b.CurZ) BETWEEN -5 AND 5;
EC    immediate
A     (AlertOperator OBJECT1; immediate; independent)
      (INSERT ON log a.itemID, XYZ; decoupled; independent)
FA    NULL
```

The above scenario is the subject of the sample run illustrated in appendix B. Some prominent features of the sample run will be highlighted here. The runs allow a trace through processing steps since the debugging information has been switched on.

The run, figure 7.5, illustrates the events which are raised when a new aircraft is being recorded, but it also shows that no knowledge was triggered.

```
New Aircraft Details

ID                  : BA747
Current Position    : 34 187 14500

Are the above details correct? (Y/N) y

AObject::putObject()

EventDetector::eventRaiseDB-Raising Object Name : BA747
EventDetector::eventRaiseDB: Raising event from BEFORE
putObject EventDetector::eventRaiseDB-Event does NOT affect
any rules - returning!
AObject::putObject-ActiveRules ... Binding TRUE

AObject::putObject-ActiveRules.... isActive TRUE
AObject::putObject-Back ActiveRules Dictionary put:
AObject::putObject-back from put ExemptRules

about to call Object::putObject(deallocate); :
AObject::putObject-about to call EventDetector-> event Raise
EventDetector::eventRaiseDB-Raising Object Name : BA747
EventDetector::eventRaiseDB: Raising event from AFTER
putObject EventDetector::eventRaiseDB-Event does NOT affect
any rules - returning!
AObject::putObject-Back from event raise:
Committing aircraft details
EventDetector::eventRaiseTrans: Raising event from BEFORE
TransCommit  EventDetector::eventRaiseTrans - Event does NOT
affect any rules - returning! EventDetector::eventRaiseTrans:
Raising event from AFTER TransCommit
EventDetector::eventRaiseTrans - Event does NOT affect any
rules - returning!
```

Figure 7.5 ATCS: Creating a new aircraft

```
Add Rule: Please enter the rules name > Avoid Aircraft
Collision
Please enter description line 1: Triggered when aircraft
movements are detected within the airspace
Please enter description line 2:
Please enter description line 3:

Please enter Event Specification: update aircraft

Please enter Condition String (if OSQL please finish with ';'
select a.Name(), b.Name() from aircraft a, aircraft b where
a.Name() = OBJECT1 and (a.CurX-b.CurX) between -5 and 5 and
(a.CurY-b.CurY) between -5 and 5 and (a.CurZ-b.CurZ) between
-5000 and 5000;

Please enter Action String, either as a SQL query of a
function call i.e. select a.ID() from aircraft a where
a.Name() = OBJECT1; please ensure to put ';' to finish
or    call AlertOperator
call AlertOperator OBJECT1
```

Figure 7.6 ATCS: Declaring a new rule

The declaration of a new rule is illustrated in figure 7.6., where after the name and description of the new rule have been captured, the triggering events are specified. In this case the event is a single internal event, which is raised when objects are put to the database. Following the specification of the triggering events, the condition or state of the database may be specified. In this case an OSQL query has been entered which tests if an aircraft which has moved within its airspace, has any possibility of being on a collision path with any other aircraft. If such a condition does arise, then the action clause is specified, in this case to call an external module which alerts the operator of the situation.

```
New Aircraft Details

ID                  : PK121
Current Position    : 29 183 19000

Are the above details correct? (Y/N) y
EventDetector::eventRaiseDB: Raising event from BEFORE
putObject Time is : Mon Jun 26 17:53:28 1995
RuleManager::knowledgeScheduler-Rule Name: Avoid Aircraft
Collision !isDisabled:1
PartCompEventSpec::OwningRule - The new Rule's name is Avoid
Aircraft Collision
RuleManager::knowledgeScheduler-Rule Name: Avoid Aircraft
Collision !isDisabled:1
PartCompEventSpec::ruleCompiledClause - Binding is TRUE
KnowlSel::testSingleEvent - but what type?
KnowlSel::testSimpleSpec - INTERNAL EVENT
KnowlSel::testSimpleSpec - INTERVALs MATCH
Clause::contextClassTypeName: aircraft
KnowlSel::testSimpleSpec - TYPES MATCH
KnowlSel::testEventSpec-after cl=rule->ruleClause(0)- IS SIMPLE
EVENT RuleManager::knowledgeScheduler - Rule Avoid Aircraft
Collision Event Specification Satisfied!
RuleManager::knowledgeScheduler before conditionStr
ConditionEvaluator::mapEventParameters--> Finished ===> About
to call ::parseQuery(select a.Name(), b.Name() from aircraft a,
aircraft b where a.Name() =  "PK121"  and (a.CurX-b.CurX)
between -5 and 5 and (a.CurY-b.CurY) between -5 and 5 and
(a.CurZ-b.CurZ) between  -5000 and 5000;)
"PK121"     "BA747"
"PK121"     "PK121"
Cardinality = 2

RuleManager::knowledgeScheduler Back from Query Evaluation,
result: 2
call AlertOperator  "PK121"   AppObject::executeCommand

ATC::AlertOperator *********  Aircraft "PK121"  in Danger Args:
"PK121"  +
```

Figure 7.7 ATCS: Trace when a rule is triggered

When the rule is triggered on the creation of a new aircraft, figure 7.7, illustrates some of the processing involved. An event is raised when the aircraft object is put to the database, which is detected by the event detector. If the event affects any rules, the knowledge scheduler is invoked. This then dispatches any affected rules to the knowledge selection module, which evaluates whether the triggered event(s) satisfy the event specification, or complete a previous part-satisfied event specification. In this case, the event is an internal event, of the right interval, and finally of the right type i.e. aircraft. Since it is a lone simple event, no further processing is necessary, the specification has been satisfied. The condition clause is then tested, which again is satisfied, since another aircraft is in the vicinity. The action clause(s) is executed, which in this case is a call to an external user application module which alerts the operator.

REFLEX allows events to be declared dynamically, as and when they are required. This is illustrated in figure 7.8 where a new event, RadarPulse, is being declared. It is simply declared by assigning it a name, description and priority (although this is not shown in the figure).

```
Add Event: Please enter the event name > RadarPulse

Please enter description line 1: Event is raised when aircraft
movement is detected
Please enter description line 2: within its airspace
Please enter description line 3:

New Event Details

Name                : RadarPulse      Num of Rules: 0
1:    Event is raised when aircraft movement is detected
2:    within its airspace
3:

Are the above details correct? (Y/N) y
```

Figure 7.8 ATCS: Declaring a new event dynamically

Once the new event has been declared, it may be utilised. The example in figure 7.9 illustrates how the existing event specification for the rule "Avoid Aircraft Collision"

was changed to incorporate the new event RadarPulse.

```
Amend Rule: Please enter rule name > Avoid Aircraft Collision
Name          : Avoid Aircraft Collision   Rule No: RM000001
Event Spec    : UPDATE aircraft

Select option     (X)Abort, (Y)Accept and Commit
            Change (E)ESL, (C)Condition, (A)Action >> e

Please enter Event Specification: event RadarPulse or after
update aircraft
Name          : Avoid Aircraft Collision   Rule No: RM000001
Description 1: Triggered when aircraft movements are
detectedwithin the airspace              2:
            3:
Event Spec    : EVENT RadarPulse OR AFTER UPDATE aircraft
Condition     : select a.Name(), b.Name() from aircraft a,
aircraft b where a.Name() = OBJECT1 and (a.CurX-b.CurX) between
-5 and 5 and (a.CurY-b.CurY) between -5 and 5 and
(a.CurZ-b.CurZ) between  -5000 and 5000;
Action:      call AlertOperator OBJECT1    Immediate   Dependent
Events        : UPDATE RadarPulse

Select option     (X)Abort, (Y)Accept and Commit
            Change (E)ESL, (C)Condition, (A)Action >> y
```

Figure 7.9 ATCS: Amending an existing ESL statement for a rule

After the rule has been modified, the next stage of the example application is to modify an aircraft object to illustrates the effect of a change to an event specification of a rule. This is illustrated in figure 7.10, where on the modification of an aircraft object, the complex event clause was triggered. This example illustrates how the complex event is processed. Even though the logical operation is a disjunction, the essentials of the process may be traced. The complex clause is broken into logical parts (clauses), These are solved individually, and then later the complete specification is tested against its logical semantics. When the event specification is satisfied, then the condition clause is tested, and if satisfied is followed by the execution of the action clause(s).

```
Amend Aircraft: Please enter aircraft name > PK121

ID                  : PK121
Current Position    : 29 183 19000
Enter the new position (Latitude Longitude Height eg 16 03 60)
33 188 19500

the X: 33 Y: 188 Z: 19500

New Aircraft Details
ID                  : PK121
Current Position    : 33 188 19500

EventDetector::eventRaiseTrans: Raising event from BEFORE
TransStart  EventDetector::eventRaiseTrans: Raising event from
AFTER TransStart  EventDetector::eventRaiseDB: Raising event
from BEFORE putObject Time is : Mon Jun 26 19:49:14 1995
RuleManager::knowledgeScheduler-Rule Name: Avoid Aircraft
Collision !isDisabled:1
KnowlSel::testEventSpec -- NEW PartCompiledEventSpec object
created KnowlSel::testSingleEvent - but what type?
KnowlSel::testEventSpec - COMPLEX EVENT, clause 0 satisfied
PartCompEventSpec::clause, index 1
KnowlSel::testSingleEvent - but what type?
KnowlSel::testSimpleSpec - INTERNAL EVENT
KnowlSel::expressionEval - Test RPN :  OR C1 C0  - length: 10
-indexPos 0
At While: OR
KnowlSel:: evalClause: OR at pos 4
KnowlSel:: evalClause Caluse OR is numbered 0
KnowlSel::testEventSpec-Complex Event Returned TRUE!  Will
return to RuleManager after delete pces
PartCompEventSpec::deleteObject
RuleManager::knowledgeScheduler - Rule Avoid Aircraft Collision
Event Specification Satisfied!
RuleManager::knowledgeScheduler before conditionStr
ConditionEvaluator::mapEventParameters--> Finished ===> About
to call ::parseQuery(select a.Name(), b.Name() from aircraft a,
aircraft b where a.Name() =  "PK121"  and (a.CurX-b.CurX)
between -5 and 5 and (a.CurY-b.CurY) between -5 and 5 and
(a.CurZ-b.CurZ) between  -5000 and 5000;)
"PK121"     "BA747"
"PK121"     "BA424"
"PK121"     "PK121"
Cardinality = 3
RuleManager::knowledgeScheduler Back from Query Evaluation,
result: 3 RuleManager::knowledgeScheduler - about to execute
Action clause call
ATC::AlertOperator *********  Aircraft "PK121"  in Danger Args:
"PK121" +
AObject::putObject-Back from event raise:
```

Figure 7.10 ATCS: Triggering a complex event specification

Figure 7.11 illustrates the granularity of event detection within REFLEX, where events are being raised on simple reads from database statements.

```
EventDetector::eventRaiseDB-Raising Object Name : BA747
EventDetector::eventRaiseDB: Raising event from AFTER read
Object EventDetector::eventRaiseDB-Event does NOT affect any
rules - returning!
ID    : BA747       Name : BA747       POS   : 34 187 14500
EventDetector::eventRaiseDB-Raising Object Name : BA424
EventDetector::eventRaiseDB: Raising event from AFTER read
Object EventDetector::eventRaiseDB-Event does NOT affect any
rules - returning!
ID    : BA424       Name : BA424       POS   : 37 190 14500
EventDetector::eventRaiseDB-Raising Object Name : PK121
EventDetector::eventRaiseDB: Raising event from AFTER read
Object EventDetector::eventRaiseDB-Event does NOT affect any
rules - returning!
ID    : PK121       Name : PK121       POS   : 29 183 19000
```

Figure 7.11 ATCS: Read events being raised

Further example rules are as follows:

- a rule to test if an aircraft arrives before its scheduled time of arrival, and to allow the aircraft to enter a new position in the landing queue, could be:

E UPDATE aircraft
C SELECT a.ID()
FROM aircraft a
WHERE a.Name() = OBJECT1
AND a.ETA() < a.STOA();
EC deferred
A (call rescheduleLandingSlot OBJECT1; deferred; independent)

- a rule to test if the weather conditions are safe for landing could be:

E UPDATE aircraft
C call isLandingConditionOK
EC deferred
A (call queueToLand OBJECT1; deferred; independent)
FA (call redirectToAlternateLandingSite OBJECT1; immediate; independent)

7.4.2. Student Records System

Many students are registered at a University. The University needs a system to maintain these many student records. These systems are generally known as Student Records Systems (SRS).

The following model, figure 7.12, will be used throughout the example.

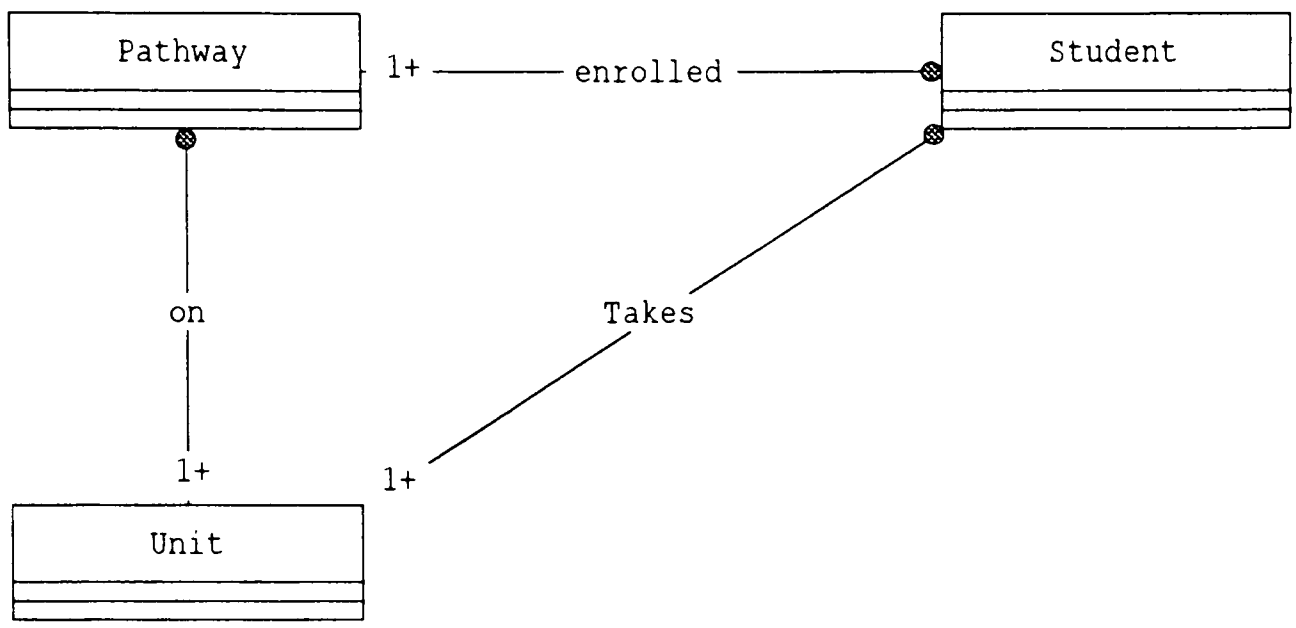


Figure 7.12 Student Records System Schema

This simply states that students are registered on Pathways (courses), and that the Pathways define which Units (modules) a student must take. The student is attached to occurrences of a given unit, i.e. the unit is the concept or metadata, the occurrence is the actual teaching session.

7.4.2.1. Traditionally

The administration systems have traditionally been a maintenance nightmare. This can be explained as follows: a particular university has many faculties, which in turn have many schools. The requirements from the SRS for each of the schools would be similar but just slightly different, i.e. a school A might want a report formatted

slightly differently to school B. It is these differences that cause the maintenance problems, since for each report a different program is required, the base program being the same but copied using 'cut & paste' techniques.

When the program is changed i.e. a bug fix, or a further requirement such as print the date and time of print request, each different version of the program must be located, edited and compiled separately.

7.4.2.2. Active Approach

This maintenance scenario can be addressed using active database techniques where the required logic is represented as rules in the database. The rules are triggered on certain situations taking place e.g. the reading of data, or the printing of a table.

Figure 7.13, illustrates the initial phase of creating a new rule, which will be triggered on the invocation of a user report. The rule will execute slightly different logic depending on the user, thus allowing logic to be maintained centrally and ultimately reducing the maintenance overhead.

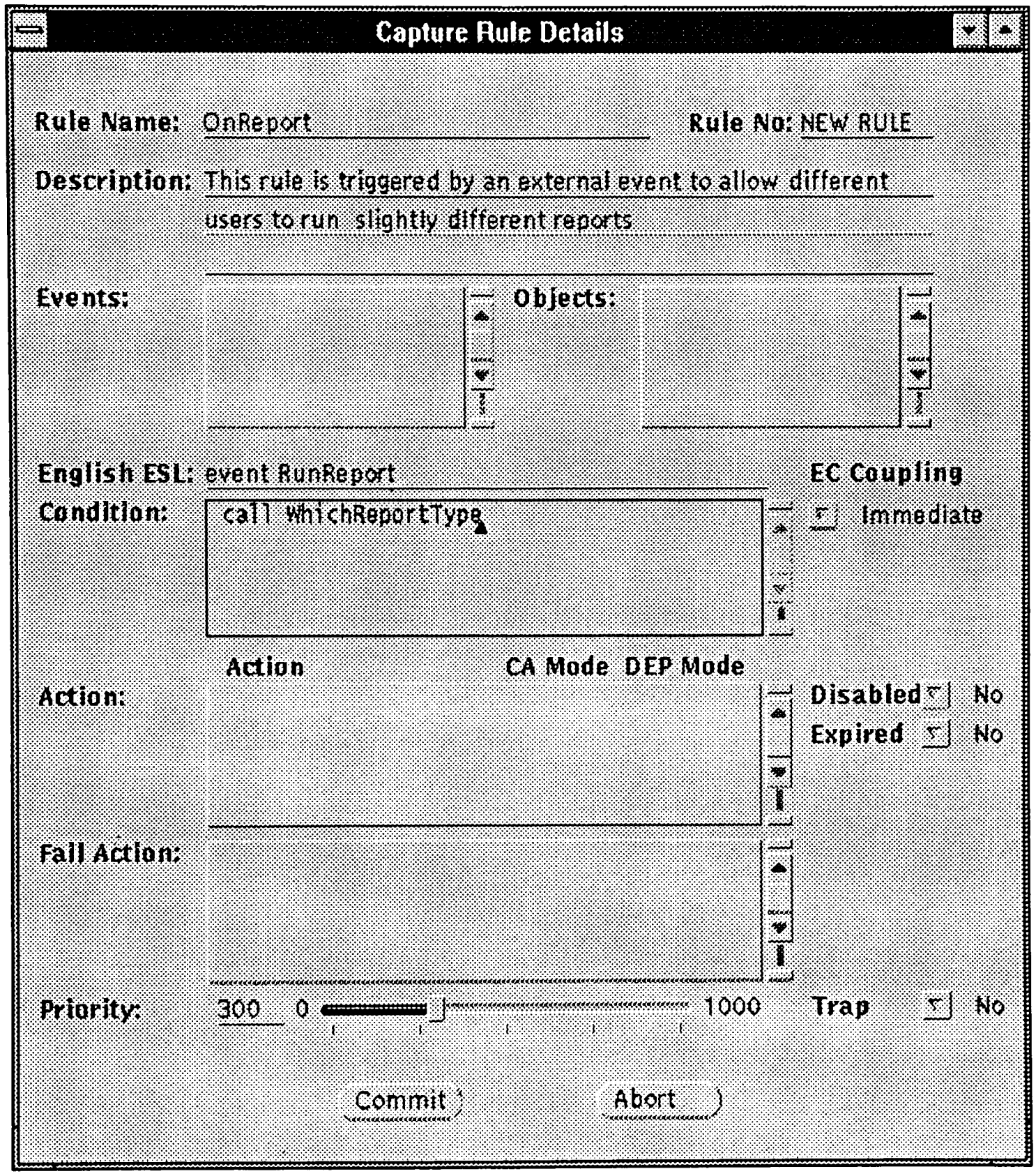


Figure 7.13 SRS: Creating a new rule

This is illustrated by the two figures, 7.14 and 7.15, where each shows an action which is marginally different, but one is action when the situation succeeds, the other when the condition fails, i.e. a fail action.

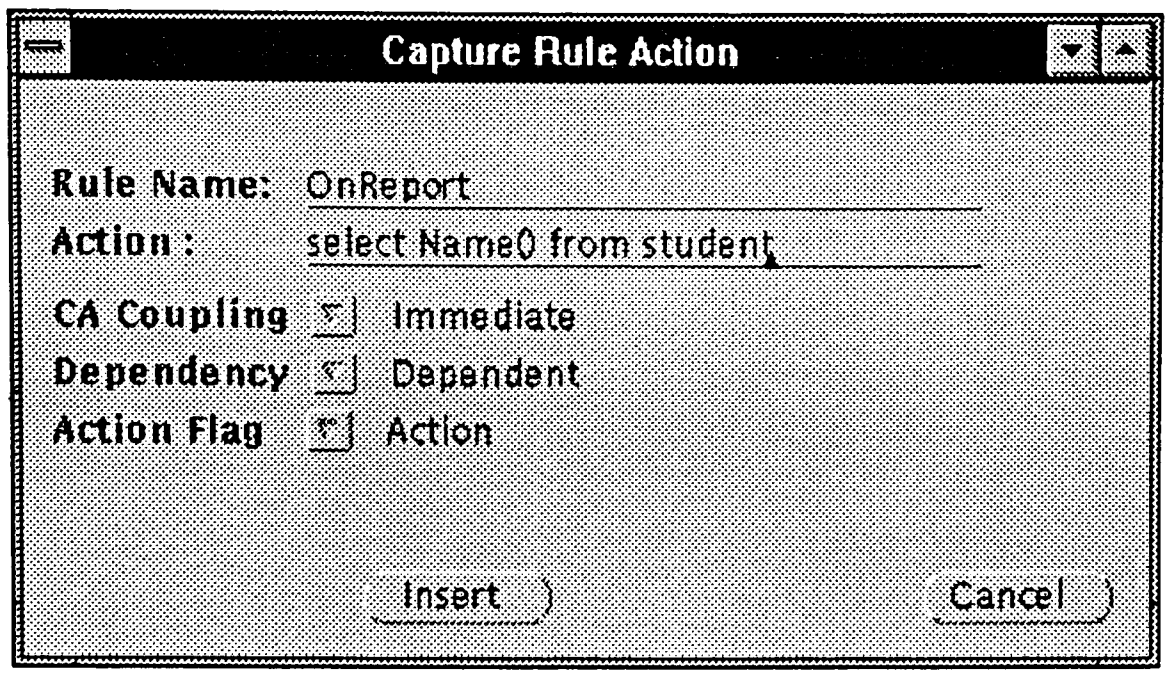


Figure 7.14 SRS: Creating a new rule action

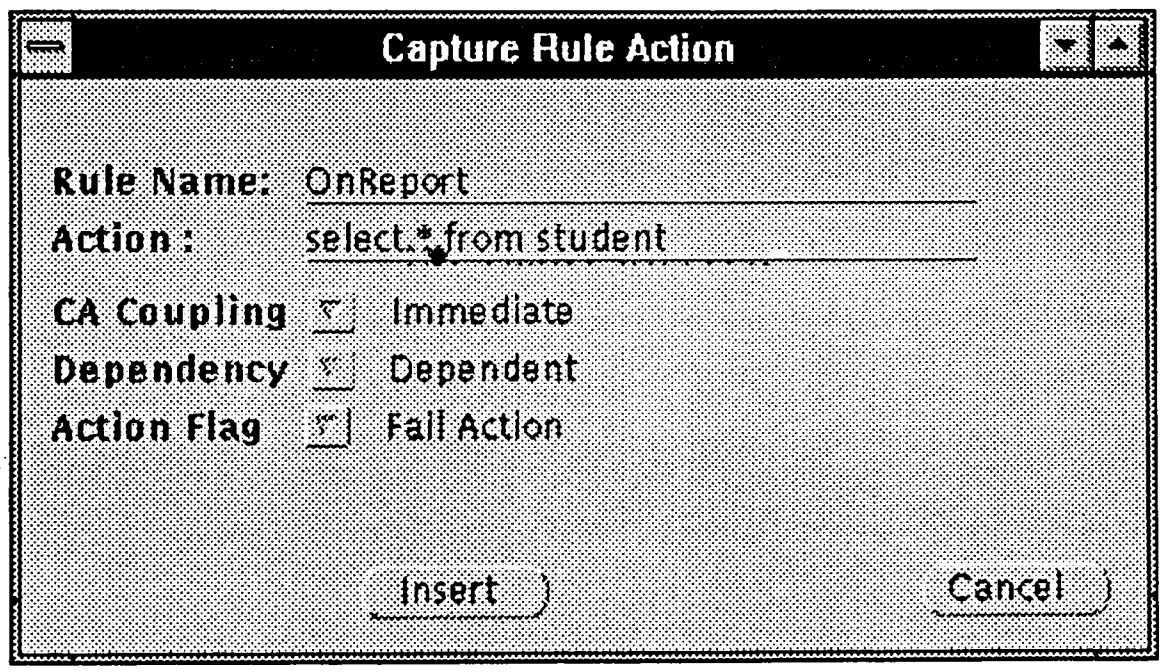


Figure 7.15 SRS: Creating a new rule fail action

Once the rule action and fail actions have been declared, the rule looks like that in figure 7.16,

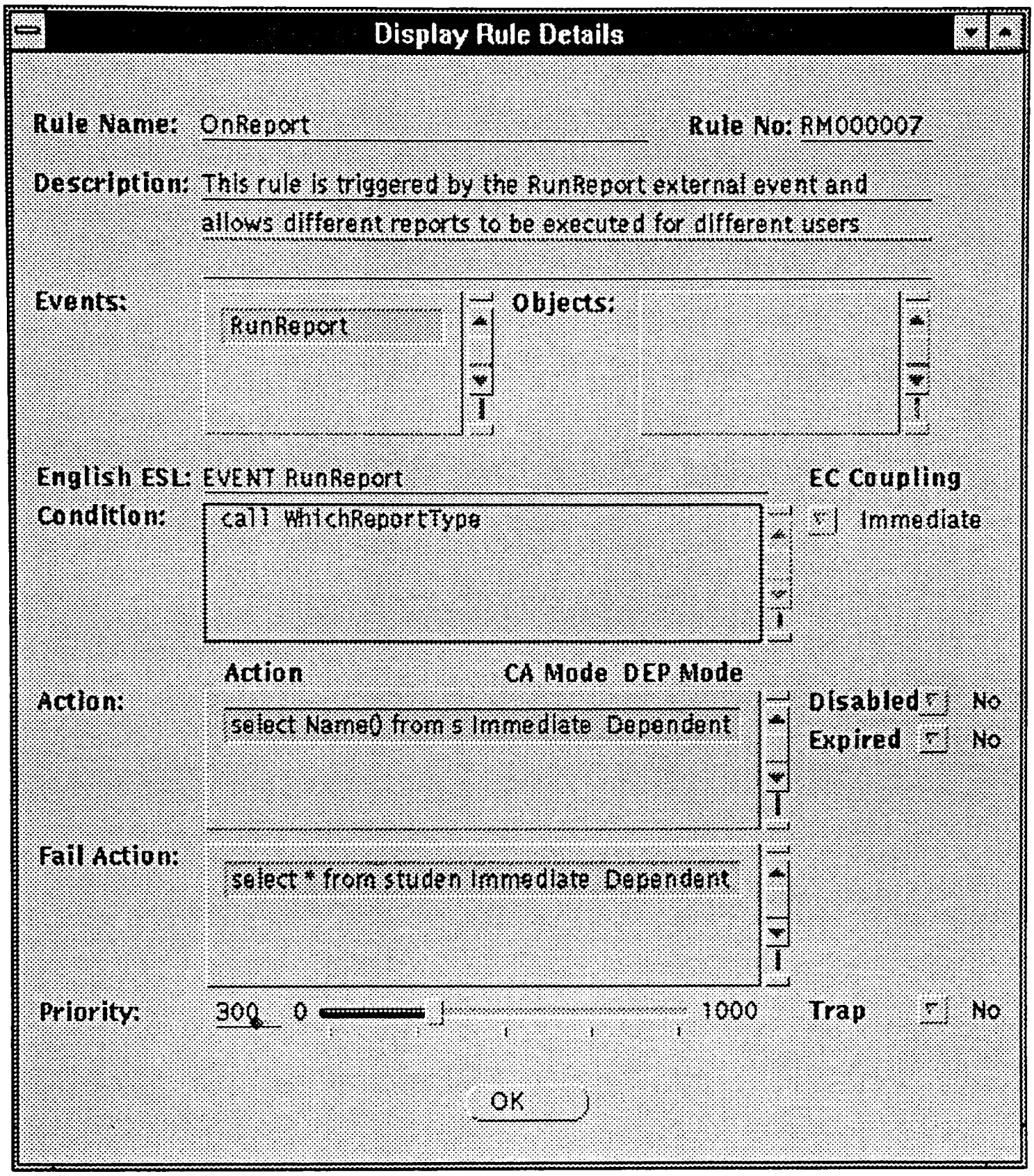


Figure 7.16 SRS: Displaying an existing rule

In the above example, Vis was used to set up the knowledge. To fire the knowledge, this has to take place within a user application because the rule is triggered by the invocation of an external event. This can be simulated by the text mode application interface, of which a sample run can be found in Appendix B.2.2. Once the knowledge is fired, and it succeeds, the rule action is executed, an abbreviated sample of the run is highlighted in figure 7.17.

```
Raise Event: Enter event name > RunReport

Argument List: Please enter any arguments (if any) > computing

KnowlSel::testEventSpec -- NEW PartCompiledEventSpec object
created
KnowlSel::testEventSpec-after cl=rule->ruleClause(0)- IS SIMPLE
EVENT
RuleManager::knowledgeScheduler - Rule OnReport Event
Specification Satisfied!
RuleManager::knowledgeScheduler before conditionStr
ConditionEvaluator::mapEventParameters--> Finished ===> About
to call ::parseQuery(call WhichReportType)
AppObject::executeCommand
AppObject::executeCommand - commandStr: call WhichReportType
<-> evArgs: computing
AppObject::executeCommand - about to switch(call
WhichReportType) -> evArgs: computing
SRS::WhichReportType    External Condition test, test for
Computing School
SRS::WhichReportType    Args: computing

ConditionEvaluator::returned from executeCommand: 1
RuleManager::knowledgeScheduler Back from Query Evaluation,
result: 1
RuleManager::knowledgeScheduler - about to execute Action
clauseselect Name() from student;
ExecutionModule::mapEventParameters--> Finished ===> About to
call ::parseQuery(select Name() from student;)

ExecutionModule::executeCommand- CommandType: select
MappedStr: select Name() from student;
EventDetector::eventRaiseDB-Raising Object Name : (null)
EventDetector::eventRaiseDB: Raising event from AFTER read
Object
EventDetector::eventRaiseDB-Event does NOT affect any rules -
returning!
"Waseem"
Cardinality = 1
Raised
```

Figure 7.17 SRS: Triggering and executing a rule action

In this example, if the user was a member of the School of Computing, only the name of all the students in the database are returned. If the user requesting the report was from another school then the fail action would be invoked, as in figure 7.18, where the fail action simply differs by requesting all of the attributes of the students.

```
AppObject::syntaxCheck commandStr: WhichReportType
AppObject::executeCommand - about to switch(call
WhichReportType) -> evArgs: Mathematics
SRS::WhichReportType    External Condition test, test for
Computing School
SRS::WhichReportType    Args: Mathematics
External Condition Fail! Non Computing School

ConditionEvaluator::returned from executeCommand: 0
RuleManager::knowledgeScheduler - about to execute Action
clauseExecutionModule::executeCommand - FailAction!
requestedselect * from student;
ExecutionModule::mapEventParameters--> Finished ===> About to
call ::parseQuery(select * from student;)

ExecutionModule::executeCommand- CommandType: select
MappedStr: select * from student;
EventDetector::eventRaiseDB-Raising Object Name : (null)
EventDetector::eventRaiseDB: Raising event from AFTER read
Object
EventDetector::eventRaiseDB-Event does NOT affect any rules -
returning!
"37133"     #1Dictionary      #2Dictionary      "Waseem"    77
      16    3     65    (charPtr*)0xa47d4 "(null)"
#3Dictionary      #4Dictionary      1342028904  (void*)0x8e0c0
      634412
Cardinality = 1
Raised
```

Figure 7.18 SRS: Triggering a rule and executing a rule fail action

The above example applications illustrated some of the features of the model and prototype. The following section describes the level of functionality attained within the prototype.

Further example SRS rules are as follows:

- a rule to automate the process of a student transferring from one course to another by setting a progress flag on the units to indicate that the student has tranfered out of them, could be:

```
E    UPDATE StudentPathway
C    SELECT      a.RefNo()
     FROM        Student a, StudentPathway b
     WHERE       b.RefNo() = OBJECT1
```

```
                AND     a.RefNo() = b.RefNo()
                AND     b.Progress() = TransferAppliedFor;
EC      deferred
A       (SELECT         a.RefNo()
        FROM            Student a, StudentPathway b, StudentUnit c
        WHERE           b.RefNo() = OBJECT1
                AND     a.RefNo() = b.RefNo()
                AND     b.Progress(Transfered)          // side affect, set value
                AND     c.Progress(Transfered);
        deferred; independent)
        (call informTargetSchool OBJECT1; deferred, dependent)
```

- a rule to enforce the constraint that a student may only take units which are applicable to the students chosen pathway, could be:

```
E       UPDATE StudentUnit
C       SELECT          a.RefNo()
        FROM            Student a, StudentPathway b, StudentUnit c, PathwayUnit d
        WHERE           c.RefNo() = OBJECT1
                AND     a.RefNo() = b.RefNo()
                AND     b.Pathway() = c.Pathway()
                AND     c.Pathway() = d.Pathway()
                AND     c.Unit()  = d.Unit();
EC      immediate
A       (Abort; immediate; dependent)
```

7.5. Functionality of Prototype

The main resultant prototype, number nine, incorporates the majority of features specified in the model. It provides a powerful platform for the development and investigation of active applications. However, some of these features are limited or not currently available.

The event specification language (ESL) for the declaration of complex events is supported in the prototype, its major functionality being provision for external, internal and temporal events, the expression algebra, and support for validity of events.

The non-destructive knowledge constructs have been provided within the prototype although they have not been exploited. This provision is to serve as the basis for future work.

The RPC concurrency model is not supported in the final prototype, since it was developed separately and due to the time constraint, has not been incorporated. This will be the subject of future work.

7.6. Summary

This chapter has served to illustrate that the REFLEX model, which had goals of providing active functionality by being adaptive and portable, evolved from its initial goals through feedback from successive prototypes, to become a comprehensive and powerful tool. The model supports many novel features which serve to increase the generality and usefulness of the active database. REFLEX not only provides active features to a host DBMS, but it provides much more functionality which augments the host DBMS. It is an extensible platform for further research into current and future DBMS issues, such as temporality.

Chapter 8

Conclusions and Future Work

The work presented in this thesis had the objective of investigating how best active knowledge management facilities could be added to existing commercial object-oriented database management systems. This chapter evaluates the results presented in the thesis, and establishes whether they have fulfilled the stated goal. Suggestions and descriptions of further work will also be made. Some of the work performed for this thesis has been published previously, and this has been listed in Appendix A.

8.1. Introduction

As highlighted by authors such as Hull and King [Hull 87] and Widom [Widom 94], the knowledge requirements of a system should more closely be represented in the database, i.e. domain knowledge and data should be integrated in some way. This work addresses this requirement. The approach adopted is that of active database systems, which are database systems that monitor situations of interest and, on their occurrence, trigger or activate an appropriate and timely response.

This work supported its major goal of determining how active functionality can be added to existing commercial DBMSs, and investigates practical considerations by the

construction of a series of working prototypes. These considerations are: (i.) determining the form of adhering to the portability and adaptability requirements, (ii.) the form the knowledge model should take, (iii.) how the events within the system are to be represented, (iv.) how triggering events are specified, (v.) how the database state is to be tested, and (vi.) how the user will interact with the system effectively.

The chapter is structured as follows: A summary of the research is presented in section 8.2. This is followed, section 8.3, by future work suggested by this research. And finally section 8.4. concludes the chapter, and indeed the thesis, with a discussion of the contribution of this work.

8.2. Summary of Research

This research concentrated on two main areas, (i.) the REFLEX experience which investigated the various design issues of an active database, and (ii.) its portability and adaptability across platforms. These are discussed in this section.

The REFLEX active extension necessarily introduces knowledge management/expert system facilities to a given DBMS. This means that there are certain components akin to those found in expert systems, within the REFLEX model, the major ones being

i. the rule/knowledge definition facility; where the elicited knowledge of an application domain is entered into the knowledge base. It differs however from the simple antecedent-consequent pair found in expert systems in that triggering events also require consideration and specifying.

ii. the knowledge management system. A management system exists which allows the creation and maintenance of rules and events dynamically. It is also responsible for the scheduling of rule evaluation and execution.

iii. the event specification system; which allows the composite events that

cause a rule to be in-context to be declared.

iv. the event detection sub-system. As events of differing types occur they must be detected and appropriate action taken.

v. knowledge evaluation system. On the raising of an event, many rules could be affected. This system determines which rules are brought into context by the raising of an event.

vi. condition evaluation system. If a rule has been brought into context, then its condition must be evaluated.

vii. rule execution system. If a rule's condition has been satisfied, then its action must be executed.

A summary of the salient parts of the research model follow.

8.2.1. Loose coupling

A major aim of the REFLEX active extension is that it is both portable and adapts to its host DBMS. A number of alternative solutions were considered, and a layered architecture approach was adopted, where a gateway or interface exists between REFLEX and any host database through which all traffic must pass. The interface is controlled by a Transparent Interface Manager (TIM), which raises any internal or temporal events. REFLEX is loosely coupled to its host DBMS, which may be interchanged with another DBMS with minimal effort. The domain knowledge is however, tightly coupled with the DBMS. Thus the resultant active system operates as if it is a bespoke active DBMS.

8.2.2. Extended ECA (EECA)

With other active models (i.e. HiPAC, Starburst, Postgres, Samos, Adam and Ode), on the satisfaction of the condition clause a specified action takes place. With REFLEX's EECA knowledge model however, not only one but multiple actions may be specified. The knowledge model attempts to minimise the problems of situation

redundancy, where the same situation may specify multiple action and fail action clauses, and extending the scope of the condition clause by allowing access to external conditions. The fail-actions, introduced by REFLEX's EECA, may be executed if the condition was not satisfied. Each of the many actions are transactions in there own right, and are expressed in tuples defining their CA coupling mode and dependency with the triggering transaction.

8.2.3. Events as first-class objects

The knowledge model supports events as first-class objects which can be created dynamically at will. A further advantage is that the events adhered to the goal of uniformity, where all components of the system are represented in the same uniform manner thus allowing maintenance tasks to be simplified. Representing events as objects also allows for optimisation, as described in the following section.

8.2.4. REFLEX Model Optimisation

Following from representing both events and rules as first-class objects, REFLEX maintains knowledge in each event as to which rule it affects i.e. each event maintains lists (indexes or pointers) to each of its rules. Thus improving its efficiency. This ability has been recognised as important since, as reported by Widom [Widom 94], low speed is the main hindrance to the application of active database technology.

8.2.5. English ESL

REFLEX supports composite events and provides a powerful language to allow the declaration of these events. The event specification language, known as English ESL, has a range of logical operators on heterogeneous event types i.e. internal, temporal and external.

The algebra is similar to other systems such as Ode and Samos, but it differs on a number of counts, (i.) it has very clear semantics, expressed in the formal temporal

algebra, (ii.) it is simple to express clear powerful expressions, using its English ESL language, and (iii.) it has a large number of operators which cover the majority of scenarios.

8.2.6. VIS

VIS, the graphical user interface to REFLEX, has been designed to allow the acquisition of knowledge in a clear, simple and intuitive manner, which enhances operator productivity and awareness. This is unlike the interfaces of other active DBMS research prototypes, such as Ode, Adam or Samos, which all use a programmatic interface which is inherently confusing and requires a large amount of programming skills.

The modular object design of VIS interfaces with, but is separate from, REFLEX and does not require information as to the internals of REFLEX or of the applications that use it. A simple user friendly design, such as VIS, takes into account many principles which are quite often incompatible i.e. flexibility and integrity, and hence can be used by both end users and application programmers to advantage.

The importance of the role that HCI principles play were recognised as significant, especially in safety critical systems, like the type that REFLEX supports e.g. ATCS.

8.2.7. Concurrency

Conventional von-Neuman architecture systems have almost reached the perimeter of current technology, in that their sequential nature limits performance. Most tasks can be divided into parallel subtasks. These subtasks can be executed concurrently in order to shorten the total time required for performing the tasks. The REFLEX model has a number of concurrency approaches, which are reviewed below.

8.2.7.1. Trap

REFLEX supports a unique mechanism to be exercised only for the most critical of

rules. On the raising of an event, the event specification clause of a rule must be evaluated, and if satisfied, its condition clause must then be evaluated. If, however, the rule is set with a trap priority, both the event specification and condition clause are evaluated concurrently.

8.2.7.2. Remote Procedure Calls (RPC)

From the outset, REFLEX was designed as a concurrent model, where many of its modules would be operating simultaneously. This was encouraged in the design by ensuring each module had a specific purpose, a defined interface and were autonomous. During the process of building a concurrent prototype using RPCs, a number of issues were discovered, (i.) using normal RPCs to distribute the modules onto the same or different physical machines, the model did not require any changes, except for the RPC calling conventions to be added. The result however, was a distributed but still essentially sequential model. Another type of RPC was required, (ii.) multi process RPC. Using this approach, multiple processes could be distributed onto various machines concurrently, but the design or calling sequence had to be essentially the opposite to that of the normal model. Where instead of the KMK calling the KSM modules, the KMK would instantiate the required number of KSM modules, which would then take control and call the KMK to request the rules to process, i.e. the KMK had become the server rather then the client.

8.2.8. Reflections on the Second Platform Implementation: POET

On first investigation it was thought that ONTOS and POET, bearing in mind that the former cost approximately 200 times more than the latter, would be vastly different in terms of functionality and interface. This was borne out since ONTOS with its impressive pedigree and maturity outclassed POET in almost every criteria except value-for-money. Even so, POET does provide most of the features you would expect to find within an object-oriented database system.

Since a goal of this research was to provide a portable and adaptive extension, the perceptions of portability and open systems were considered, i.e. if a programming language claims to be ANSI standard, then the same program should compile and execute without modification under the same standard on different platforms.

To port and adapt REFLEX to the POET platform did not require a change of design. But the porting process did however challenge the preconceptions that a system will port simply because it is of a given type. This is because there are (i.) terminology problems where homonyms exist, e.g. set manipulation in ONTOS and POET (ii.) false standards, such as ANSI compliant products and (iii.) expected features of a product are missing, e.g. the type manager in POET. In reality the porting process was not as straightforward as originally anticipated, since each product had its quirks. The elapsed time during the porting process was significant, since time was required to investigate the operation of the new platform and any lack of functionality. The actual time expended porting to the platform was minimal i.e. the order of approximately one week.

The user interface VIS, was not ported from the X/Open Motif environment to that of Microsoft Windows. This was not seen as necessary since the prototype worked against the database in text mode and because of the learning curve involved, and lack of the time resource.

8.2.9. Novel Active Applications

The REFLEX prototype has allowed investigation into new paradigms with some novel approaches to existing research problems.

8.2.9.1. Cortextual Parser

The REFLEX active database has introduced the concept of active algorithms, where the database learns over time how to dynamically tune and mutate an algorithm. The

initial investigation was conducted in the image processing domain to produce segmentation algorithms for robotic vision systems [Naqvi 94b]. The model was named the cortextual parser after the cortex of the eye.

8.2.9.2. Dynamic Active Schema Integration Model (DASIM)

There is a need to integrate schemas in any large organisation, but in a federated environment this becomes even more important. Especially if the component schemas are continually changing themselves. The REFLEX database was to investigate this domain, and produce a dynamic active schema integration model, called DASIM [Naqvi 94a], which would monitor component schemas for change, and then pass any amended schemas to ISIT for evaluation.

8.3. Future Directions

This research has highlighted further areas that require investigation.

8.3.1. Real data trials

There has been much research into the field of active database management, but up until now the work has primarily been based at the pure theoretical issues such as how can we make an active database. More important issues such as how will an active database behave in the real world, have not been tackled apart from superficial example applications. This scenario has arisen because the field is relatively immature, but now that real working active databases are emerging from the research labs, we are in a position to investigate issues such what kind of knowledge will need to be stored, what sort of application areas are appropriate, what are the performance characteristics/profiles over use, how can these be changed.

8.3.2. Temporal extensions

The ability to store information/history about an application area, and query any of the

historic states is akin with the way humans think. It is time that database systems reflect that view.

The REFLEX model has introduced the concept of non-destructive knowledge, where a rule may be entered, but may not be changed after it has been fired once. After that point, a new rule must be created to reflect any amendments with a pointer to the new rule from the old terminated rule.

The REFLEX model could be extended with temporal facilities so that it provides a non-temporal database with temporal facilities at the core/engine level, in addition to active facilities.

8.3.3. Optimisation and parallelism

Further work is required into making the active extension faster. Speeding up the entire database is not possible since accessing the internals of the host is not an option within this research.

Alternative architectures could be developed where the REFLEX extension could exist on a separate system to the host, for example as a main memory system encompassing the entire knowledge/rule base in its cache, or a system of multiple parallel processors.

8.3.4. Petri net compiler

The REFLEX model was originally specified and modelled using Petri-nets. Further work into Petri-nets could involve the development of a run-time Petri-net interpreter of complex events.

8.3.5. VIS Extensions

Interaction interfaces between users and the system, are probably the area of a system that cause the most inefficiencies. VIS, as REFLEX's graphical user interface, provides an easy, intuitive approach to interfacing with the complexities of domain knowledge

management. It does require further enhancements, for instance (i.) in-depth analysis tools for the knowledge model i.e. static and dynamic checking of the knowledge base, (ii.) predictive facilities to allow the event specification and more importantly the database state tests to be automatically formed, and (iii.) VIS also be ported to the alternate MS Windows platform.

8.3.6. Analysis and Design of Rules

Tools are required which will alleviate the problem of actually acquiring and analysing business rules and formulating them into EECA rules. This requires a high level abstracted approach, where the business or organisation rules are entered and these are automatically broken down into smaller useable, more formal rules i.e. a 4GL approach to knowledge requirements akin to those found for vertical markets.

8.4. Conclusions and Contributions

This research has shown how active functionality can be added to an existing commercial database, in a portable, adaptive, and efficient manner. This was proven by two implementations on different platforms, namely ONTOS and POET. The preconceptions which maintain that porting a product is a straightforward task if you abide by so called open systems standards, were proved wishful. From this experience, if you wish to create a portable system, one must be cautious and challenge every assumption.

After surveying other related work it was discovered that other research groups were tackling the domain of active databases from a proprietary outlook i.e. they developed their models and prototypes as stand-alone systems. This research differs from the related work initially because of its loose-coupling to its underlying commercial database technology unlike other research groups which have not taken the approach that activity could be ported and adapted to a given host. Most research groups have

used in-house database technology where the source code was available, such as Starburst, Postgres, ADAM, ODE and ETM. In some cases they have licensed the source, such as SAMOS with its use of ObjectStore, even so the resultant systems are tightly coupled to their underlying hosts. The approach taken by this research means that active functionality may be augmented to an existing commercial object DBMS, providing benefits to the organisation, such as support of legacy systems and minimal training costs.

During the research, ideas were readily mapped to working prototypes which allowed concepts to be evaluated as to their usefulness and efficiency, quickly. This approach of using prototypes proved successful and allowed contributions to the domain of active databases to be made.

The primary contributions of this thesis are the REFLEX model with its Extended ECA knowledge model, its uniform object model, its English ESL with its clear semantics, its novel index links to all related objects, its non-destructive knowledge model, its distribution model, and its graphical user interface.

Not only has REFLEX allowed the provision of new knowledge, but it is also a powerful vehicle for future knowledge discovery since it may be used for research into active applications, and specifically data collection. Even with the handful of rules embodied in the example applications, a lot of activity takes place within the database on an event being raised, and hence the behaviour of systems which require large amounts of rules need to be analysed.

Chapter 9

Bibliographic References

[Abiteboul 87]

Abitebout S. and Hull R., "IFO: A Formal Semantic Database Model", ACM Transactions on Database Systems, Vol 12 No 4, pp 525-565, 1987

[Agarwal 92]

Agarwal R. and Tanniru M., "A Petri-net based approach for verifying the integrity of production systems", International Journal of Man-Machine Studies, Vol. 36 No 3 pp 447-468, March 1992

[Agrawal 89]

Agrawal R. and Gehani N.H., "Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++", 2nd Int. Workshop on Database Programming Languages, Portland, OR, June 1989

[Agrawal 90]

Agrawal R., Gehani N.H. and Srinivasan J., "Odeview: The Graphical Interface to Ode", Proc. 1990 ACM SIGMOD Intl. Conf. on Management of Data, Atlantic City, NJ, May 1990

[Aiken 92]

Aiken A., Widom J. and Hellerstein J.M., "Behaviour of Database Production Rules: Termination, Confluence, and Observable Determinism", Proc. 1992 ACM SIGMOD Intl. Conf. on Management of Data

[Albano 89]

Albano A., "Conceptual Languages: A Comparison of ADAPLEX, Galileo and Taxis", in (Eds) Schmidt J.W. and Thanos C., "Foundations of Knowledge Base Management", pp 395-408, Springer-Verlag, 1989

[Allen 81]

Allen J.F., "An interval-based representation of temporal knowledge", Proc. of the 7th IJCAI, 1981

[Allen 84]

Allen J.F., "Towards a General Theory of Action and Time", Artificial Intelligence, Vol 23, No 2, 1984

[Al-Zobaidie 87]

Al-Zobaidie A. and Grimson J.B., "Expert systems and database systems: how they can serve each other?", Knowledge Engineering, Vol 4 No 1, February 1987

[Al-Zobaidie 88]

Al-Zobaidie A. and Grimson J.B., "Use of metadata to drive the interaction between database and expert systems", Information and Software Technology, Vol 30 No 8, October 1988

[Andrews 87]

Andrews T. and Harris C., "Combining Language and Database Advances in an Object-Oriented Developmet Environment", OOPSLA, 1987

[Annett 71]

Annette J., Duncan K.D., Stammers R.B., and Gray M.J., "Task analysis", Training Information No 6, HMSO, London 1971

[Astrahan 79]

Astrahan M.M., Blasgen M.W., Chamberlain D.D., Gray J.N., King W.F., Lindsay B.G., Lorie R.A., Mehl J.W., Price T.G., Putzolu G.R., Schkolnick M., Selinger P.G., Slutz D.R., Strong H.R., Tiberio P., Traiger I.L., Wade B.W., and Yost R.A., "System R: A Relational Database Management System", Computer, Vol. 12 No. 5, pp 43-48, May 1979

[Baer 70]

Baer J., Bovet D. and Estrin G., "Legality and Other Properties of Graph Models of Computations", Journal of the ACM, Vol. 17, No. 3., July 1970, pp. 543-554

[Beeri 91]

Beeri C. and Milo T., "A Model for Active Object Oriented Database", Proc. of the 17th Int. Conf. on Very Large Data Bases, Barcelona, Spain, 1991

[Bell 90]

Bell D.A., Shao J. And Hull M.E.C., "Integrated Deductive Database System Implementation: A Systematic Study", The Computer Journal, Vol:.33, No 1, pp 40-48, 1990

[Bell 92]

Bell D. and Grimson J, "Distributed Database Systems", Addison-Wesley, 1992

[Benmaiza 91]

Benmaiza M. and Elkaraksy, M.R., "Knowledge-based approach to Petri nets analysis", Knowledge-Based Systems Vol: 4 Iss: 3, pp. 144-56, Sept. 1991

[Bernardinello 92]

Bernardinello L. and De Cindio F., "A Survey of Basic Net Models and Modular Net Classes", Advances in Petri-nets 1992, Springer-Verlag, Lecture Notes in Computer Science, 609

[Berthomieu 91]

Berthomieu B. and Diaz M., "Modeling and Verification of Time Dependent Systems Using Petri Nets", IEEE Transactions on Software Engineering, Vol 7, No 3, March 1991

[Beynon-Davis 91]

Beynon-Davis P., "Expert Database Systems A Gentle Introduction", McGraw-Hill, 1991

[Bowers 93]

Bowers D.S., From Data to Database, 2nd Edition, Chapman & Hall, 1993

[Brenner 91]

Brenner E., Grabner J., Moosburger M., Otschko G., Schlögl K., Seifter P., Song J., Steger Ch. and Weiss R., "Design and Implementation of a Distributed Real-Time Expert-System for Fault Diagnosis in Modular Manufacturing Systems", Microprocessing & Microprogramming, Vol. 32 No 1-5 pp 799-806, August 1991

[Brodie 84]

Brodie M.L. and Ridjanovic D., "On the Design and specification of Database Transactions", in (Eds) Brodie M.L., Mylopoulos J. and Schimidt J.W., "On Conceptual Modeling", Springer-Verlag, 1984

[Brodie 87]

Brodie M.L. and Manola F., "Database Management: A Survey" in Readings in Artificial Intelligence and Databases, Morgan Kaufmann

[Brodie 93]

Brodie M.L., "Interoperable Information Systems: Motivations, Challenges, Approaches, and Status", Tutorial Notes, VLDB 93, Dublin, Ireland, August 1993

[Brownston 85]

Brownston L., Farrell R., Kent E. and Martin N., "Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming", Addison-Wesley, 1985

[Cardenas 91]

Cardenas A., and McLeod, D. (Eds.) Research Directions in Object-Oriented and Semantic Database Systems, Prentice-Hall series in DKBS, Englewood Cliffs, N.J. 1991.

[Cattell 91]

Cattell R.G.G, "Object Data Management: Object-Oriented and Extended Relational Database Systems", Addison-Wesley, 1991.

[Cercone 87]

Cercone N., and MaCalla G., "The Knowledge Frontier: Essays in the Representation of Knowledge", Springer-Verlag, New York, 1987

[Chakravarthy 89]

Chakravarthy S., Blaustein B., et al, "HiPAC: A Research Project in Active, Time-Constrained Database Management", Final Technical Report, Xerox Advanced Information Technology Division, July 1989

[Chakravarthy 90a]

Chakravarthy S., Minker J. and Grant J., "Logic Based Approach to Semantic Query Optimization", ACM Trans. on Database Systems, Vol 15 No 2, 1990

[Chakravarthy 90b]

Chakravarthy S. and Nesson S., "Making an Object-Oriented DBMS Active:

Design, Implementation and Evaluation of a Prototype", Proc. Int. Conf. Extending Database Technology, Venice, March 90

[Chakravarthy 91]

Chakravarthy S., and Misra D., "An event specification language (snoop) for active databases and its direction", Tech. Report UF-CIS-TR-91-23, University of Florida, 1991

[Chakravarthy 93]

Chakravarthy S., Anwar E. and Maugis L., "Design and Implementation of Active Capability for an Object-Oriented Database", Tech. Report UF-CIS-TR-93-001, University of Florida, 1993

[Cookney 94]

Cookney D. and Naqvi W., "Project Management for Object-Oriented Systems Development", Proc. of the 7th Int. Conf. on Systems Research, Informatics and Cybernetics, Baden-Baden, August 1994

[Copeland 84]

Copeland G. and Maier D., "Making Smalltalk a database system", Proc. SIGMOD, 1984

[Dayal 88]

Dayal U., Blaustein B., et al, "The HiPAC Project: Combining Active Databases and Timing Constraints", ACM Sigmod Record, Vol. 17, No. 1, March 1988

[Dayal 89]

Dayal U., "Active Database Management Systems", Sigmod Record, Vol. 18, No. 3, 1989

[Diaper 89]

Diaper D. and Johnson P., "Task analysis for knowledge descriptions: theory and application in training", in Cognitive Ergonomics and Human Computer Interaction, (Eds.) Long J. and Whitefield A., 1989

[Diaz 90]

Diaz O. and Gray P.M.D., "Semantic-rich User-defined Relationship as a Main Constructor in Object Oriented Database", Proc. of the IFIP TC2 Conf. on Object-Oriented Databases: Analysis, Design and Construction (DS-4), 1990

[Diaz 91a]

Diaz O. and Paton N. W., "Sharing behaviour in an object-oriented database using a rule-based mechanism", Proc. of the 9th British National Conference On Databases, Wolverhampton, 1991

[Diaz 91b]

Diaz O., Paton N. and Gray P., "A Rule Management in Object Oriented Databases: A Uniform Approach", Proc. of the 17th Int. Conf. on Very Large Data Bases, Barcelona, Spain 1991

[Dittrich 86]

Dittrich K.R., Kotz A.M. and Mulle J.A., "An Event/Trigger Mechanism to Enforce Complex Consistency Constraints in Design Databases", ACM Sigmod Record, Vol. 15, No. 3, September 1986

[Dittrich 91]

Dittrich K. and Dayal U., "Active Database Systems", Tutorial Notes, VLDB 91, Barcelona, Spain, September 1991

[Elmasri 94]

Elmasri R. and Navathe S., Fundamentals of Database Systems, 2nd Edition, Adison Wesley, 1994

[Eswaran 76]

Eswaran K.P., "Specifications, implementations and interactions of a trigger subsystem in an integrated database system", IBM Res. Rep. RJ1820, August 1976

[Forgy 82]

Forgy C., "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", Artificial Intelligence, 19, pp 17-37, 1982

[Freundlich 90]

Freundlich Y., "Knowledge Bases and Databases: Converging Technologies, Diverging Interests", IEEE Computer, November 1990

[Fry 76]

Fry J. and Sibley E., "Evolution of Data-Base Managemnet Systems", ACM Computing Surveys, Vol 8 No 1, March 1976

[Galton 87]

Galton A. (Ed), "Temporal Logics and their Applications", Academic Press, 1987

[Gatziu 93]

Gatziu S. and Dittrich K.R., "Events in an Active Object-Oriented Database System", Proc. of 1st Int. Workshop. on Rules in Database Systems, Edinburgh, September 1993

[Gehani 91]

Gehani N.H. and Jagadish H.V., "Ode as an Active Database: Constraints and Triggers", Proc. of the 17th Int. Conf. on Very Large Data Bases, Barcelona, Spain 1991

[Gehani 92a]

Gehani N.H., Jagadish H.V. and Shmueli O., "Event Specification in an Active Object-Oriented Database", Proc. 1992 ACM SIGMOD Intl. Conf. on Management of Data

[Gehani 92b]

Gehani N.H., Jagadish H.V. and Shmueli O., "Composite Event Specification in Active Databases: Model & Implementation", Proceedings of the 18th Int. Conf. on Very Large Data Bases, Vancouver, Canada, 1992

[Giarratno 89]

Giarratano, J. and Riley, G., "Expert Systems: Principles and Programming", PWS-Kent Publishing Co., Boston, 1989

[Goldberg 81]

Goldberg A., "Introducing the Smalltalk-80 System", Byte, Vol. 6 No. 8, August 1981

[Gupta 91]

Gupta R. and Horowitz E. (eds), Object-Oriented Databases with Applications to CASE, Networks and VLSI CAD, Prentice-Hall series in DKBS, Englewood Cliffs, N.J. 1991.

[Gray 93]

Gray J. and Reuter A., "Transaction Processing: Concepts and Techniques", Morgan Kaufman, San Mateo, 1993.

[GWB 92]

Gwb, "P.O.E.T. Reference Manual v.1"

[Hammer 81]

Hammer M. and McLeod D., "Database description with SDM: A semantic database model" in ACM Transactions on Database Systems, vol 6, No 3, 1981, pp 351-386

[Hanson 92]

Hanson E.N., "Rule Condition Testing and Action Execution in Ariel", Proc. 1992 ACM SIGMOD Intl. Conf. on Management of Data

[Hauser 92]

Hauser C.A., "Constructs for Modeling Information Systems with Petri Nets", Application and Theory of Petri Nets 1992, Springer-Verlag, Lecture Notes in Computer Science, 616

[Hildebrand 89]

Hildebrand T. and Treves N.,"S-CORT: A method for the development of Electronic Payment Systems", Advances in Petri Nets 1989, Springer-Verlag, Lecture Notes in Computer Science, 424

[Hillion 89]

Hillion H. P.,"Timed Petri Nets and Application to Multi-Stage Production Systems", Advances in Petri Nets 1989, Springer-Verlag, Lecture Notes in Computer Science, 424

[Hsu 88]

Hsu M., Ladin R., McCarthy D., "n xecution Model for Active Data Base Management Systems", Proceedings of the 3rd International Conference on Data and Knowledge Bases", June 1988

[Hull 87]

Hull R. and King R., "Semantic Database Modeling: Survey, Applications, and Research Issues" in ACM Computing Surveys, vol 19, No 3, 1987, pp 201-260

[Ibrahim 95]

Ibrahim M.T., "A note on the concepts of Data, Information and Knowledge", Database Systems Research Laboratory, Technical Report TR-CIS-DB0195,

University of Greenwich, 1995

[Interbase 90]

InterBase Software Corporation, "Interbase 3.0 DDL Reference Manual", 1990

[Jafar 93]

Jafar, M., Bahill, T.A, "Interactive Verification of Knowledge-Based Systems", IEEE Expert, Vol.8 No1, Feb. 1993.

[Jagannathan 88]

Jagannathan D., Guck R.L., Fritchman B.L., Thompson F.P. and Tolbert D.M., "SIM: A database system based on the semantic data model", in Proceedings of ACM SIGMOD, pp 46-55, 1988

[Jensen 90]

Jensen K., "Coloured Petri Nets: A High Level Language for System Design and Analysis", Advances in Petri Nets 1990, Springer-Verlag, Lecture Notes in Computer Science, 483

[Jianjun 92]

Jianjun Y., Feng Z., Jiati D. and Chuaijun C., "Intelligent Manufacturing cell controller IMCC-E", Human Aspects in Computer Integrated Manufacturing Conf., Tokyo, Japan, IFIP Transaction B, Vol. B-3 pp. 745-755, 1992

[Johnson 92]

Johnson P., "Human Computer Interaction: Psychology, Task Analysis and Software Engineering", McGraw-Hill, 1922

[Joseph 91]

Joseph J.V., Thatte S.M., Thompson C.W. and Wells D.L., "Object-Oriented Databases: Design and Implementation" in Proceedings of the IEEE, vol 79, No 1, 1991, pp 41-64

[Kernighan 78]

Kernighan B.W. and Ritchie D.M., "The C Programming Language", Prentice-Hall, 1978

[Khosafian 86]

Khosafian S.N. and Copeland G.P., "Object Identity", Proc. OOPSLA, 1986

[Kim 89]

Kim W., "A Model of Queries for Object-Oriented Databases" in Proceedings of the Fifth International Conference on Very Large Data Bases, 1989, pp 423-432

[Kim 90]

Kim W., "Object-Oriented Databases: Definition and Research Directions" in IEEE Transactions on Knowledge and Data Engineering, vol 2, No 3, 1990 pp 327-341

[Kim 93]

Kim W., "Object-Oriented Database Systems: Promises, Reality, and Future", in Proceedings of the 19th International Conference on Very Large Data Bases, Dublin, 1993, pp 676-687

[Kim 95]

Modern Database Systems: The Object Model, Interoperability, and Beyond, Kim W. (Ed.), Addison-Wesley, 1995

[King 84]

King R. and McLeod D., "A Unified Model and Methodology for Conceptual Database Design", in (Eds) Brodie M.L., Mylopoulos J. and Schimidt J.W., "On Conceptual Modeling", Springer-Verlag, 1984

[Kingston 87]

Kingston J., "Technical overview of Knowledge Craft", AIAI, Vol 3 No 6, 1987

[Knight 92a]

Knight B. and Ma J., "A General Temporal Model Supporting Duration Reasoning", AI Communication Journal, Vol. 5, No. 2, 1992

[Knight 92b]

Knight B., "A Deductive Approach to Temporal Databases", The Computer Journal, Vol. 35, pp A395-A402, 1992

[Knight 94]

Knight B. and Ma J., "Time representation: A taxonomy of temporal models", Artificial Intelligence Review, Vol 7, pp 401-419, 1994

[Kotz 88]

Kotz A.M., Dittrich K.R. and Mulle J.A., "Supporting Semantic Rules by a Generalized Event/Trigger Mechanism", Proceedings of EDBT, pp 76-91, 1988

[Kowalski 86]

Kowalski R.A. and Sergot M.J., "A Logic-Based Calculus of Events", New Generation Computing, No 4, pp 67-95, 1986

[Kowalski 92]

Kowalski R., "Database updates in the event calculus", Journal of Logic Programming, pp 121-146, 1992

[Leslie 95]

Leslie R., "Simulation of load-sharing alogorithms", Tech. Report CIS-COMM029501, University of Greenwich, February, 1995

[Liebowitz 86]

Liebowitz, "Useful Approach for Evaluating Expert Systems", Expert Systems, Vol. 3 No 2, 1986.

[Lindsay 80]

Lindsay B.G., Selinger P.G., Galtieri C., Gray J.N., Lorie R.A., Price T.G., Putzolu G.R., Traiger I.L., and Wade B.W., "Notes on Distributed Databases", in "Distributed Data Bases", Draffen I.W. and Poole F. (Eds.), Cambridge University Press, England, 1980

[Ling 92]

Ling D.H.O. and Bell D.A., "Modelling and Managing Time in Database Systems", The Computer Journal, Vol. 35, No. 4, 1992

[Lohman 91]

Lohman G.M., Lindsay B., Pirahesh H. and Schiefer K.B., "Extensions To STARBURST: Objects, Types, Functions, and Rules", CACM October 1991, Vol 34, No 10

[Löwgren 93]

Löwgren J., "Human-computer interaction", Studentlitteratur, Lund, 1993

[Lui 91]

Lui Lung-Chun, Horowitz E., "Object Database Support for Case", Object-Oriented Databases with Applications to CASE, Networks and VLSI CAD, Prentice-Hall series in DKBS, Englewood Cliffs, N.J. 1991.

[Luger 89]

Luger G.F. and Stubblefield W.A., "Artificial Intelligence and the Desisn of Expert Systems", Benjamin Cummings

[Manola 86]

Manola F. and Dayal U., "PDM: An Object-Oriented Data Model", Proceedings 1st International Workshop on Object-Oriented Database Systems, September 1986

[Marsan 89]

Marsan M. Ajmone, "Stochastic Petri Nets: An Elementry Introduction", Advances in Petri Nets 1989, Springer-Verlag, Lecture Notes in Computer Science, 424

[May 89]

May R. and Shepherd R., "Occam and the Transputer", Advances in Petri Nets 1989, Springer-Verlag, Lecture Notes in Computer Science, 424

[McCarthy 63]

McCarthy J.,"Situation, Actions, and Causal Laws", Memo 2, Standford Artificial Intelligence Project, 1963

[McCarthy 69]

McCarthy J. and Hayes P.J., "Some Philosophical Problems from the Standpoint of Artificial Intelligence", (eds.) Meltzer B. and Michie D., Machine Intelligence 4, Edinburgh, 1969

[McCarthy 89]

McCarthy D.R. and Dayal U.,"The Architecture of an Active Data Base Management System", Proc. ACM SIGMOD Intl. Conf. on Management of Data, Portland, June 1989

[McDermott 81]

McDermott J., "R1: the formative years", AI Magazine, Summer, 1981

[McDermott 82]

McDermott D.V., "A Temporal Logic for Reasonong about Processes and Plans", Cognative Science, Vol 6, 1982

[Medeiros 90]

Medeiros C.B. and Pfeffer P., "A Mechanism for Managing Rules in an Object-

oriented Database", Altair Technical Report, 1990

[Meseguer 92]

Meseguer J., Montanari U. and Sassone V., "On the Semantics of Petri Nets", Concur '92, 1992, Springer-Verlag, Lecture Notes in Computer Science, 630

[Moat 94]

Moat A. and Naqvi W., "PETENG and the Modeling of Petri Nets", Proc. of the 7th Int. Conf. on Systems Research, Informatics and Cybernetics, Baden-Baden, August 1994

[Minsky 75]

Minsky M., "A Framework for Representing Knowledge", The Psycology of Computer Vision, McGraw-Hill, 1975, pp. 211-277

[Mylopoulos 86]

Mylopoulos J. And Wong H.K.T., "Some features of the TAXIS Data Model", in Proceedings of the 6th International Conference on Very Large Data Bases, Montreal, Canada, 1986

[Mylopoulos 90]

Mylopoulos M., "Object Orientation and Knowledge Base Management", in Object-Oiented Databases: Analysis, Design and Construction (DS-4), Meersman R.A., Kent W. and Khosla S. (Eds.), IFIP, North-Holland, 1991

[Naqvi 92]

Naqvi W. and Ibrahim M.T., "The REFLEX Active Database System", Database Systems Research Laboratory, Technical Report TR-CIT-DB0692, University of Greenwich, 1992

[Naqvi 93a]

Naqvi W. and Ibrahim M.T., "REFLEX: An Active Database Extension", poster at the 11th British National Conference on Databases, Keele, July, 1993

[Naqvi 93b]

Naqvi W. and Ibrahim M.T., "REFLEX Active Database Model: Application of Petri-Nets", Proc. of the 4th Int. Conf. on Database and Expert Systems Applications, Prague, September 1993

[Naqvi 93c]

Naqvi W. and Ibrahim M.T., "REFLEX: An Active Object-Oriented Database Model", Database Systems Research Laboratory, Technical Report TR-CIT-DB0493, University of Greenwich, April, 1993

[Naqvi 93d]

Naqvi W. and Ibrahim M.T., "Rule and Knowledge Management in an Active Database System", Proc. of 1st Int. Workshop. on Rules in Database Systems, Edinburgh, September 1993

[Naqvi 94a]

Naqvi W., Hughes C., Ibrahim M.T. and Al-Zobaidie A., "Active Schema Integration", Proc. of the 7th Int. Conf. on Systems Research, Informatics and Cybernetics, Baden-Baden, August 1994

[Naqvi 94b]

Naqvi W. and Panyiotou, "Active Databases and Evolving Algorithms for Image Processing", Proc. of the 3rd Pacific Rim Int. Conf. on Artificial Intelligence, Beijing, August 1994

[Naqvi 94c]

Naqvi W. and Ibrahim M.T., "Active Databases and Extending their Knowledge Model", Proc. of the 7th Int. Conf. on Systems Research, Informatics and Cybernetics, Baden-Baden, August 1994

[Naqvi 94d]

Naqvi W. and Ibrahim M.T., "EECA: An Active Knowledge Model", Proc. of the 5th Int. Conf. on Database and Expert Systems Applications, Athens, September 1994

[Naqvi 95a]

Naqvi W. and Ibrahim M.T., "Active Distribution by Stealth", Proc. of the 6th Int. Conf. on Database and Expert Systems Applications, London, September 1995

[Naqvi 95b]

Naqvi W. and Panyiotou S., "Applied Active Databases for Evolving Image Processing Algorithms", Proc. of the 6th Int. Conf. on Database and Expert Systems Applications, London, September 1995

[Navathe 92]

Navathe S.B., Tanaka A., Chakravarthy S., "Active Database Modelling and Design Tools: Issues, Approach, and Architecture", IEEE Bulletin of the TC on Data Engineering, Vol 15 (1-4), 1992

[Nierstrasz 89]

Nierstrasz O., "A Survey of Object-Oriented Concepts", in *Object-Oriented Concepts, Databases and Applications*, (Eds.) Kim W and Lochovsky F., ACM Press and Addison-Wesley, 1989

[O'leary 90]

O'leary, T.J., Goul, M., Moffit, K.E., Essam Radwan, A., "Validating Expert Systems", IEEE Expert, Vol. 5 No3, June 1990.

[ONTOS 91]

"ONTOS Reference Manual", ONTOS Inc, 1991

[Paton 89]

Paton N.W., "ADAM: An Object-Oriented Database System Implemented In Prolog", Proc. of the 7th British National Conference On Databases, 1989

[Paton 90]

Paton N. and Diaz O., "Metaclasses in obkect-oriented databases", Proc. of the IFIP TC2 Conf. on Object-Oriented Databases: Analysis, Design and Construction (DS-4), 1990

[Paton 91]

Paton N. and Diaz O., "Object-oriented databases and frame-based systems: comparison", Information and Software Technology, Vol 33 No 5, June 1991

[Patterson 77]

Paterson J.L., "Petri Nets", ACM Computing Surveys, Vol. 9, No. 3, September 1977

[Patterson 81]

Paterson J.L., "Petri Net Theory and the modeling of Systems", Prentice-Hall, 1981

[Peckham 88]

Peckham J. and Maryansk F., "Semantic Data Models" in ACM Computing Surveys, Vol 20, No 3, 1988, pp 153-189

[Perdu 91]

Perdu D.M. and Levis A.H., "A Petri Net Model for Evaluation of Expert Systems in Organisations", Automatica, Vol. 27, No. 2, pp. 225-237, 1991

[Prock 91]

Prock J., "A New Technique for Fault Detection Using Petri Nets", Automatica, Vol. 27, No. 2, pp. 239-245, 1991

[Randell 94]

Randell M.J., "Parallelisation of an Active Database System", Tech. Report CIS-DSRL049401, University of Greenwich, April 1994

[Ringland 87]

Ringland G., "Structured Object Representation - Schemata and Frames", Approaches to Knowledge Representation, (Eds.) Ringland and Duce, 1987, pp 81-99

[Rumbaugh 91]

Rumbaugh J., Blaha M., Premerlani W., Eddy F. and Lorensen W., "Object-Oriented Modeling and Design", Prentice-Hall International, 1991

[Schek 91]

Schek H. and Scholl M.H., "From Relations and Nested Relations to Object Models", Proc. of the 9th British National Conference on Databases, Wolverhampton, July 1991

[Schreier 91]

Schreier U., Pirahesh H., Agrawal R. and Mohan C., "Alert: An architecure for transforming a passive DBMS into an active DBMS", Proc. of the 17th Int. Conf. on Very Large Data Bases, Barcelona, Spain, 1991

[Silva 89]

Silva M. and Valette R., "Petri Nets and Flexible Manufacturing", Advances in Petri Nets 1989, Springer-Verlag, Lecture Notes in Computer Science, 424

[Shipman 81]

Shipman D.W., "The Functional Data Model and the Language DAPLEX" in ACM Transactions on Database Systems, vol 6, No 1, 1981, pp 140-173

[Smith 77]

Smith J.M. and Smith D.C.P., "Database abstractions: Aggregation and generalization", ACM Transactions on Database Systems, Vol 2 No 2, March 1977

[Soloway 87]

Soloway E., Bachant J. and Jensen K., "Assessing the maintainability of XCON-in-RIME: Coping with the problems of a very large rule base", Proc. AAAI-87, Los Altos, CA, 1987

[Stevens 92]

Stevens W.R., "Advanced Programming in the UNIX Environment", Addison Wesley, 1992

[Stonebraker 87]

Stonebraker M., "The Design of the POSTGRES Storage System", Proc. of the 13th Int. Conf. on Very Large Data Bases, Brighton, England 1987

[Stonebraker 88]

Stonebraker M., "Future Trends in Database Systems", Proc. of the Fourth Int. Conf. on Data Engineering, 1988

[Stonebraker 89a]

Stonebraker M. and Neuhold E., "The Laguna Beech Report", International Institute of Computer Science, Technical Report #1, Berkeley, June 1989.

[Stonebraker 89b]

Stonebraker M., Hearst M. and Potamianos S., "A Commentary on the POSTGRES Rules System", Sigmod Record, Vol. 18, No. 3, September 1989

[Stonebraker 90]

Stonebraker M., et. al, "Third Generation database system manifesto", in Object Oriented Analysis, Design and Construction, Kent W. and Meersman R. (Eds.), 1990

[Stonebraker 91a]

Stonebraker M., "Manageing persistent objects in a multi-level store", Proc. of the 1991 ACM SIGMOD Conf. on Management of Data, Denver, May 1991

[Stonebraker 91b]

Stonebraker M. and Kemnitz G., "The POSTGRES Next-Generation Database

Management System", CACM October 1991, Vol 34, No 10

[Stonebraker 93]

Stonebraker M., Agrawal R., Dayal U., Neuhold E.J. and Reuter A., "DBMS Research at a crossroads: The Vienna Update", Proc. of the 19th Int. Conf. on Very Large Data Bases, Dublin, Ireland 1993

[Stroustrap 86]

Stroustrup B., "The C++ Programming Language", Addison Wesley, 1986

[Sun 93]

Sun Microsystems Inc, "SunNet Manager 2.1 Reference Manual", 1993

[Sybase 90]

Sybase Corporation, "Sybase Reference Manual", 1990

[Taubner 88]

Taubner D., "On the implementation of Petri Nets", Advances in Petri Nets 1988, Springer-Verlag, Lecture Notes in Computer Science, 340

[Tsichritzis 78]

Tsichritzis D. and Klug A. (Eds.), "The ANSI/X3/SPARC DBMS Framework", AFIPS Press, 1978

[Widom 89]

Widom J. and Finkelstein S. J., "A Syntax and Semantics for Set-Oriented Production Rules in Relational Databse Systems", Sigmod Record, Vol. 18, No. 3, September 1989

[Widom 90]

Widom J. and Finkelstein S. J., "Set-Oriented Production Rules in Relational Databse Systems", Proc. of the 1990 ACM SIGMOD Conf. on Management of Data, Denver, Atlantic City, New Jersey, May 1990

[Widom 91]

Widom J., Cochrane R.J. and Lindsay B.G., "Implementing Set-Oriented Production Rules as an Extension to Starburst", Proc. of the 17th Int. Conf. on Very Large Data Bases, Barcelona, Spain, 1991

[Widom 93]

Widom J., "Deductive and Active Databases: Two Paradigms or Ends of a

APPENDIX A

Author's Related Publications

The author's related publications are itemised below, followed by the full text for the first five of the listed papers.

Naqvi W. and Ibrahim M.T., "Active Distribution by Stealth", Proceedings of the Workshop at the 6th International Conference on Database and Expert Systems Applications, London, September 1995

Naqvi W. and Ibrahim M.T., "EECA: An Active Knowledge Model", Proceedings of the 5th International Conference on Database and Expert Systems Applications, Athens, September 1994

Naqvi W. and Ibrahim M.T., "REFLEX Active Database Model: Application of Petri-Nets", Proceedings of the 4th International Conference on Database and Expert Systems Applications, Prague, September 1993

Naqvi W. and Ibrahim M.T., "Rule and Knowledge Management in an Active Database System", Proceedings of 1st International Workshop. on Rules in Database Systems, Edinburgh, September 1993

Naqvi W. and Panyiotou, "Applied Active Databases for Evolving Image Processing Algorithms", Proceedings of the 6th International Conference on Database and Expert Systems Applications, London, September 1995

Naqvi W. and Ibrahim M.T., "Active Databases and Extending their Knowledge Model", Proceedings of the 7th International Conference on Systems Research, Informatics and Cybernetics, Baden-Baden, August 1994

Naqvi W. and Panyiotou, "Active Databases and Evolving Algorithms for Image Processing", Proceedings of the 3rd Pacific Rim International Conference on Artificial Intelligence, Beijing, August 1994

Naqvi W., Hughes C., Ibrahim M.T. and Al-Zobaidie A., "Active Schema Integration", Proceedings of the 7th International Conference on Systems Research, Informatics and Cybernetics, Baden-Baden, August 1994

Moat A. and Naqvi W., "PETENG and the Modeling of Petri Nets", Proceedings of the 7th International Conference on Systems Research, Informatics and Cybernetics, Baden-Baden, August 1994

Naqvi W. and Ibrahim M.T., "REFLEX: An Active Database Extension", poster at the 11th British National Conference on Databases, Keele, July, 1993

Active Distribution by Stealth

Waseem Naqvi and Mohamed T.Ibrahim

School of Computing and Mathematical Sciences
University of Greenwich, London, SE18 6PF, U.K.
{w.naqvi, m.ibrahim}@greenwich.ac.uk
http://www.gre.ac.uk/~nw01/reflex

Abstract

Databases and especially active databases which not only maintain data but also the domain knowledge, require evermore powerful machines and processing power. This research addresses this issue by optimising and concentrating the processing power available within the environment and making it available to an application. This paper reports on the method employed within the REFLEX distribution model to effectively steal cycles.

Keywords: REFLEX, active database, stealing cycles, load sharing, distributed database, EECA, cycle stealing, client-server, rpc

1. Introduction

There are presently a number of active database prototypes, including for example HiPAC [2], Starburst [6], ADAM [3] and REFLEX [10]. These databases, which react automatically to a given situation, may prove useful for the encoding of general domain knowledge for an application. The inclusion of knowledge, however, which must be processed within the database incurs overhead. Recently, this prompted a panel of active database experts to state that the performance of these types of databases is not yet sufficient to support mission-critical applications, as reported by Widom [18].

Whilst working on improving the performance of the REFLEX active database system, by the use of innovative indexes [10] and allowing many parts of the architecture to operate concurrently an interesting observation was made. This may be stated as follows: in a given organisation there may be many computers which may be of the same type i.e. many PC's and/or SUN's, which are networked together in some way. Some of these computers may be used by development staff (i.e. intensively used), some by operational staff (i.e. average usage) and others could be on the desks of management and administration staff and are generally used to read email messages and perhaps a little wordprocessing or spreadsheet work (i.e. little used in terms of CPU time). Hence a situation exists where many of the computers across an organisation are simply not being used to their full potential. This research attempts to utilise this situation by determining where spare CPU cycles are available and distributing the work/load to those machines. Effectively getting closer to the goal of truly transparent distributed systems.

The paper is structured as follows. The REFLEX active database system is briefly introduced with respect to the interaction between certain modules. This is followed, in section three, by a summary of the results of an analysis of organisational machines, i.e. which machines are being under utilised and algorithms for how the target machine for off-loading work can be selected. The distribution of REFLEX is addressed in section four. Finally section five concludes the paper and highlights the results of the distribution.

2. REFLEX

REFLEX is a portable and adaptive active database extension for a given commercial database. Currently it has been implemented on top of ONTOS [13] and POET [14] object databases. The former on the Sun Solaris and XWindows platform and the latter on the PC MS-Windows 3.1 environment. Unlike the traditional ECA knowledge model, as described by McCarthy and Dayal [7], which most active databases employ i.e. HiPAC and Starburst, REFLEX has an extended knowledge model the EECA [10], which addresses the problems associated with situation redundancy such as the multiple declaration of similar rules.

REFLEX and its architecture has been reported on before [8, 10], but will be briefly summarised. REFLEX like HiPAC and Ode [4], supports complex events, which may affect many rules. As the events are raised they are signalled to the Event Manager from three sources (i.) internal events by the Transparent Interface Manager (ii.) external events by the application programs and (iii.) temporal events by the system clock notifier unit. The Event Manager is responsible for the logging of the events and their notification to the Knowledge Management Kernel (KMK), which evaluates whether the event affects any rules. If the event affects any rules, the rules in question are passed to the Knowledge Selection Module (KSM), which evaluates whether the rule's event specification clause has been satisfied. If it has been satisfied, i.e. it is '*in context*', then the rule is returned to the KMK ready for its condition clause to be tested. The KMK passes the rule to the Condition Evaluation Module (CEM) which tests the condition clause. If the clause is satisfied, the CEM returns the rule with a status of '*fireable*'. The KMK then passes the rule to the Execution Supervisor, which then executes the actions. Each of the afore mentioned components are modeled and implemented as objects.

Clearly a substantial bottle-neck could occur at the Knowledge Selection Module, which ascertains whether the event has actually satisfied a rule, or caused a previously part-satisfied rule to become fully satisfied, since for each rule that an event could affect, the KSM must be called.

This research investigates the optimisation via distribution of this KSM module. The following section introduces the distribution method and algorithms proposed to locate a suitable target for the distribution.

3. Distribution

Distributed database systems (DDBS), which are a union of a database system and a communications network, have had a goal of being transparent. So much so that Walker and Popek state "As much as possible, the location of your

data, the allocation of resources, and even the existence of a network should be hidden from you" [17]. In the context of their LOCUS distributed system architecture, Walker and Popek have identified six types of transparency of which this research is interested in the following two i.) Performance transparency, which means that the overhead referencing involved in remote resources is so small it is negligible, and ii.) Process transparency, where each process should see all the other processes as if they were executing on the same machine. This research adheres to these goals, where some parts of an application run on one machine but may also run on many other machines simultaneously. An additional goal of this research, is that the system will select the target machine by virtue of its effective utilisation i.e. is it presently being used or not.

A mechanism for distributing a logical application across a network in the client-server paradigm is the Remote Procedure Call (RPC) [1]. Where one process (the caller or client process) can have another process (the server process) execute a procedure call, as if the caller process had executed the procedure call in its own address space. Because the client and the server are now two separate processes, they no longer have to live on the same physical machine.

Before distributing a process, the status of the target machines needs to be ascertained. This can be accomplished in many ways, using established network management tools such as SunNet Manager [19], or some rather more obvious system utilities such as vmstat (reports certain statistics kept about process, virtual memory, disk, trap and CPU activity) and rup (time up and load average).

The status of machines on the network was detected by setting up a shell script which simply called the rup function at regular periods for a few sample weeks, and wrote the results to a file. The survey was attempting to establish how often the machines are loaded, and by how many runnable processes. Conversely the communications overhead to send the message had to be considered, for this the network delay could be detected by using the simple ping function i.e. how long would a machine take to simply respond to say it was alive.

This was conducted on a sample set of 20 machines of various CPU power ratings. Full in depth results can be found in [5]. A summary of the results indicated that communications overhead ranged from 30ms to 60ms. But importantly, in order to select a machine with enough spare capacity, every machine in the sample set did not require testing. For the sample set of 20 machines, testing just three machines produced a reasonable target host. Although frequently the target was selected on the very first random testing, since on the first test there was always enough spare capacity. This can be extrapolated and implies that if there were 50 machines in the sample, a target could again be selected by random testing of just five machines.

The following section examines the distribution and parallelism of the REFLEX model.

4. The REFLEX Distribution Model

In order to overcome the performance bottle-neck of requiring every rule that an event affects to be processed by the KSM, the tasks of the KSM either had to be optimised and made more efficient, or the KSM duplicated for each affected rule. The

approach taken within this work was to parallelise REFLEX and specifically the KSM. The current local procedure mode of interaction between the KMK and the KSM, as illustrated in figure 1, can be described as follows. The KMK (or client) calls the KSM object (or server) and passes its arguments i.e. the event and affected rule. The KSM then takes control, carries out its processing, and eventually returns back control. At which point the results of the procedure are extracted and the caller continues execution. These steps are repeated for each affected rule.

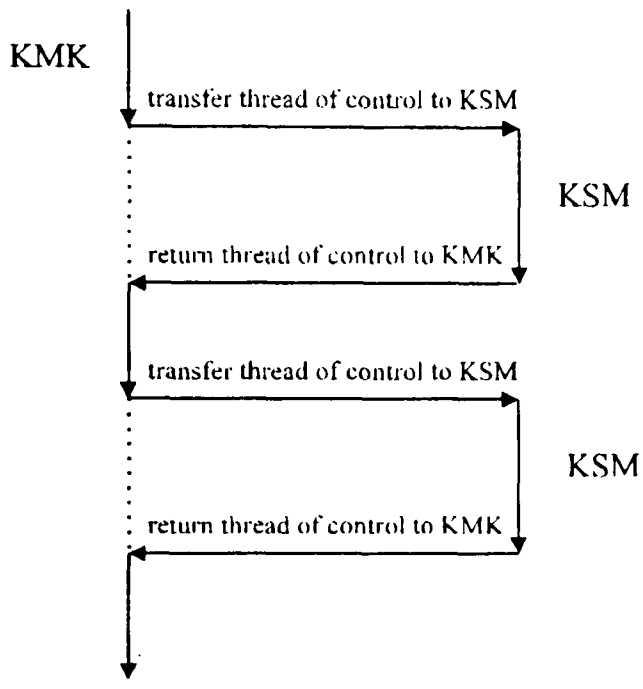


Figure 1 Existing Sequential Model

The first stage at reducing the overhead caused by the KSM was to parallelise the module on the same host machine. There are many approaches that could have been taken to parallelise the module (e.g. parallel architecture database machines), but since REFLEX is a portable active database extension, most were ruled out as they could not be made portable. The adopted approach was to instantiate many KSM processes within the same machine using RPC since this method would be both upgradeable and portable.

The RPC approach is similar to the local procedure call, figure 1, in that one thread of control logically winds through two processes, one is the caller's process, the KMK, the other is a server process, the KSM, and waits for a reply message. The call message contains the procedure's parameters. The reply message contains the

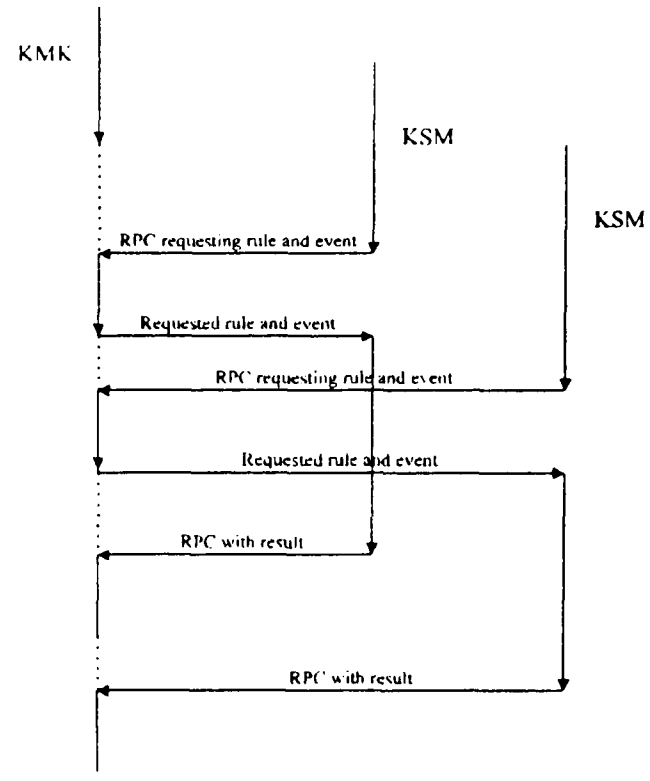


Figure 2 RPC Concurrency Model

procedure's results and once received the results are extracted and the caller's execution is resumed. This approach does not solve the problem of single-tasking sequential execution of the KSM since the thread of control may only be active in one process at a time, therefore only one of the two processes is active at any given time. The RPC protocol does, however, allow for multi-threaded control where it is possible for the calling process to do useful work while waiting for a reply from the server. But this again is not a real solution since the only useful work the KMK would be doing is making more calls to the KSM. The KSM would then buffer these calls and execute them sequentially.

In order to achieve the desired parallelism, the

REFLEX architecture implementation was *reengineerd to work in reverse* [12]. Rather than making a call to the KSM server from the KMK client, the reverse scenario was required i.e. make the KSM the calling process and the KMK the server. This way, multiple instances of the KSM, running concurrently, can call the KMK, as illustrated in figure 2.

The KMK instantiates a number of copies of the KSM that are required and then waits to service each instance of the KSM. Each KSM calls the KMK for the rule and event data, which it tests and calls the KMK with the result. Once all of the KSMs have called the KMK with the results of their tests, the KMK stops acting as a server and moves on to its next task.

4.1. Implementation Details

The reimplementation involved a complete rewrite of the KSM so that it would initiate server requests and transfer data by means of RPCs. The KMK, unlike the KSM, is a large management object and hence only parts of it required changing (those that interacted with the KSM), i.e. new XDR (eXternal Data Representation, RPC implementation independent data types) data types were introduced along with handle information i.e. program number, version and procedure numbers. The main method within the KMK which was responsible for scheduling the KSMs, *knowledgeScheduler()*, had to be completely rewritten. A KSM had to be instantiated for each affected rule, the name of the host and rule number that the KSM is to test are passed in the form of command-line arguments. The KMK would then continue its processing. When all of the KSMs have completed by calling the RETRESULT procedure, the knowledgeScheduler() loops through the results of the rules and acts upon those that were in-context. RETRESULT gets the process number and the result of the test, which it stores in the variable *result*, indexed by rule number. The procedure also increments the global variable *processCount*. It is this variable that the KMK tests to determine when to stop servicing the KSMs. Interested readers may retrieve a copy of the source code both before and after it was changed to use multi-threaded RPC, by accessing the REFLEX web page.

4.2. Distributing the KSM

The next stage is to steal unused cycles throughout an organisational network and to distribute the KSM to machines which are being predominantly underused. The previous section has described how the KMK-KSM coupling was parallelised using multi-threaded RPCs. Hence the KSM can have multiple instantiations on the same host machine or on different machines throughout the network. The question at this stage is how do we select the machine to which the KSM can be dispatched. REFLEX assumes the machines in an organisation which can participate in client-server sharing of an active application are homogeneous and that they are networked transparently. In the case of this research the machines are all Sun workstations connected via a tcp/ip network using the Sun NFS transparent filing system, ensuring that each machine maps to the same required database area, figure 3.

As discussed in section three, evaluating whether a particular machine has spare capacity

incurs an overhead of between 30ms to 60ms. Hence it is only reasonable to distribute the KSM if the required processing is perceived to take longer than the communications overhead, for instance 100ms. This was taken into account by only distributing the KSM if the signalled event affected more than one rule and for each rule that the event specification clause referenced more than one event i.e. complex events.

If the KMK establishes that the event affects many rules, it informs the Communications Manager (CM). The Idling Sensor (part of the CM) randomly tests the status of some sample machines until it finds a target with enough spare capacity and a low number of running processes. As noted in section three, the number of *hits* required to select a machine is low i.e. even if 50 machines are in the sample set only five random tests should be required provide a suitable target. Once a target is selected, the Module Dispatcher, informs the KMK of the target workstation, which then instantiates a KSM on the target machine in the manner described in section 4.1. This is continued until all of the rules are being processed by their own respective KSM instantiations, and until the KMK has received a RETRESULT from each of the KSMs.

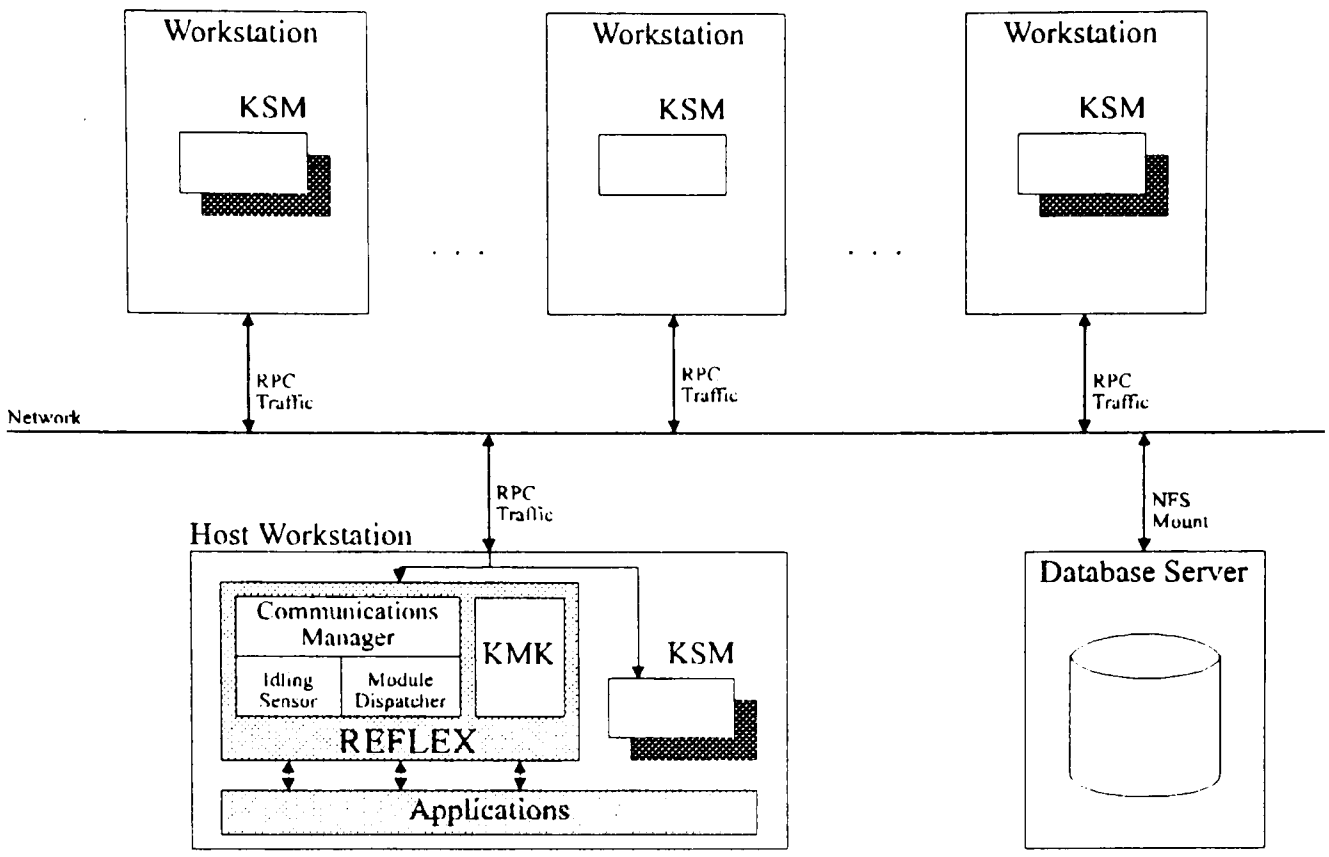


Figure 3 REFLEX with Distributed Knowledge Selection Module

5. Conclusions and Further Work

Performance is a crucial feature for active database systems. This position was articulated by concenus of a group of active database experts [18]. This paper has presented a method of optimising an active database system by parallelising certain parts, presently only the Knowledge Selection Module (KSM), and distributing the work-load to a reasonably under-utilised machine. Effectively stealing CPU cycles from machines with plenty to spare.

In order to parallelise the REFLEX active database model, to operate within the client-server paradigm, the design had to be effectively *'turned on its head'*, with the original client, the Knowledge Management Kernel, becoming the server to the also role reversed KSM client.

The stealth approach although piloted in the context of active database research can be applied to any domain where increases in processing power are required, and where spare processing capacity seems to be available albeit in a distributed form within many organisations. It could be the incarnation of the phrase *'the whole is greater than the sum of the parts'*.

A number of issues were not addressed in the current prototype, and we hope to resolve them in the near future. For instance, authentication. In the described model, when the KSM called the KMK it was not required to identify itself. Clearly, due to the nature of the data being passed between processes, stronger security is required where a client is forced to identify itself to the server, ensuring that the client is not an imposter trying to steal or corrupt data. Another issue is that of fault tolerance, the current model does not provide any recovery procedures in the case of problems in the network. If for some reason, one of the instances of the KSM does not call the RETRESULT procedure, the KMK will hang, waiting to service the KSM. Some form of time out and retransmission policy must be implemented.

Other modules such as the CEM and Execution Supervisor will also require parallelising and distribution, once further results and data i.e. benchmarking, have been collected from the existing prototype.

6. References

[1] Birrell A.D., Nelson B.J., "Implementing Remote Procedure Calls", ACM Trans. on Computer Systems, Vol. 2 No. 1, February 1984

[2] Chakravarthy S., Blaustein B., Buchmann A. et al, "HiPAC: A Research Project in Active, Time-Constrained Database Management", Final Technical Report, Xerox Advanced Information Technology Division, July 1989

[5] Diaz O., Paton N. and Gray P., "A Rule Management in Object Oriented Databases: A Uniform Approach", Proc. of the 17th Int. Conf. on Very Large Data Bases, Barcelona, Spain 1991

[4] Gehani N.H., Jagadish H.V. and Shmueli O., "Event Specification in an Active Object-Oriented Database", Proc. 1992 ACM SIGMOD Intl. Conf. on Management of Data

[5] Leslie R., "Simulation of load-sharing alogorithms", Tech. Report CIS-COMM029501, University of Greenwich, February, 1995

[6] Lohman G. M., Lindsay B., Pirahesh H. and Schiefer K. B., "Extensions To STARBURST: Objects, Types, Functions, and Rules", CACM October 1991, Vol 34, No 10

[7] McCarthy D.R. and Dayal U.,"The Architecture of an Active Data Base Management System", Proc. ACM SIGMOD Intl. Conf. on Management of Data, Portland, June 1989

[8] Naqvi W. and Ibrahim M.T., "REFLEX Active Database Model: Application of Petri-Nets", Proc. of the 4th Int. Conf. on Database and Expert Systems Applications, Prague, pp 233-240, September 1993

[9] Naqvi W. and Ibrahim M.T., "Rule and Knowledge Management in an Active Database System", Proc. of 1st Int. Workshop. on Rules in Database Systems, Edinburgh, pp 58-69, September 1993

[10] Naqvi W. and Ibrahim M.T., "EECA: An Active Knowledge Model", Proc. of the 5th Int. Conf.

on Database and Expert Systems Applications, Athens, pp 380-389, September, 1994

[11] Naqvi W. and Panyiotou S., "Applied Active Databases for Evolving Image Processing Algorithms", Proc. of the 6th Int. Conf. on Database and Expert Systems Applications, London, September, 1995

[12] Naqvi W. and Randell M.J., "Parallelisation of an Active Database System", Tech. Report CIS-DSRL049401, University of Greenwich, June, 1994

[13] "ONTOS Reference Manual", ONTOS Inc, 1991

[14] "POET 2.1 Programmer's & Reference Guide", POET Software Corporation, 1994

[15] "Network Programming Guide", Sun Microsystems, Inc, 1990

[16] Stonebraker M. and Kemnitz G., "The POSTGRES Next-Generation Database Management System", CACM October 1991, Vol 34, No 10

[17] Walker B.J. and Popek G.J., "A Transparent Environment", Byte, July 1989

[18] Widom J., "Research Issues in Active Database Systems: Report from the Closing Panel at RIDE-ADS'94", Sigmod Record, Vol 23 No 3, September 94

[19] "SunNet Manager Programmers Guide", Sun Microsystems Inc, 1993

EECA: An Active Knowledge Model

Waseem Naqvi and Mohamed T. Ibrahim

Database Systems Research Laboratory
University of Greenwich, London, SE18 6PF, U.K.
{w.naqvi, m.ibrahim}@greenwich.ac.uk

Abstract

General purpose triggers are central to active database management systems, along with knowledge in the form of production rules. The predominant knowledge model is based on Event-Condition-Action (ECA) triples. Our research has found this model to be limiting and inefficient in both operation and declaration clarity as it causes unnecessary replication of rules. An extension is proposed to the ECA knowledge model to permit a semantically concise and precise declaration of the knowledge. This extension (EECA) has been integrated into the REFLEX active database prototype. This paper reports on the EECA model and gives an overview of the REFLEX model, its architecture and novel features.

Keywords: knowledge model, active database, event specification, object-orientated, EECA

1. Introduction

Conventional databases are *passive* repositories of data where actions are preformed by either user or explicit program requests. In these databases, data is separated from its meaning or semantics. However, this situation is changing and nowadays commercial database management systems provide support for integrity constraints. This support is provided by a simple collection of triggers which are usually associated with specific data objects and are not held in one place. However, there is much more domain knowledge that an application designer would like to support.

Additionally, in traditional database systems, data management is separated from the application's processing logic. The domain knowledge is hidden in and distributed across the application's code. In an active database system, data, knowledge and parts of the processing logic relating to events and conditions that require action are under the control of an *active* Database Management System (ADBMS).

What distinguishes an active database from a conventional database is that the former is enhanced with *active* behaviour. It automatically responds to internal or external events (or changes in the environment). The domain knowledge is usually captured in the form of Event-Condition-Action (ECA) triples as reported in [MD89]. When an event occurs which causes a change in the environment provided that the specified

conditions are satisfied then the stated action(s) of one or more rules are automatically performed by the system without user intervention.

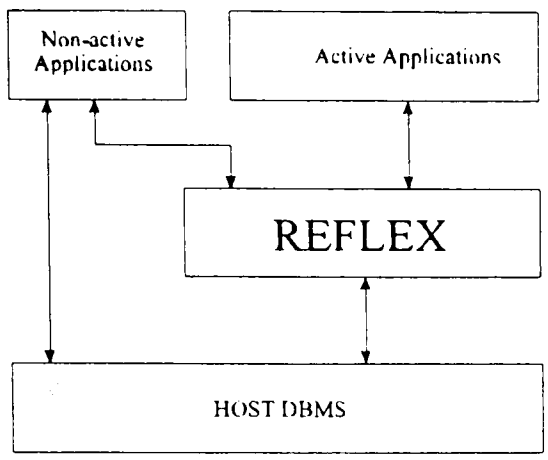


Figure 1 REFLEX Extension

The main aim of this research is to provide a flexible, portable *active database facility* for existing systems. A research prototype, known as REFLEX, has been developed which allows the host database to respond immediately to changes in the internal state of the database or its environment. It must respond within a specific time-constraint i.e. reflexively.

This paper reports on extensions to the ECA model, provided by the REFLEX active database prototype. The paper is organised as follows. An overview of REFLEX is presented in section two, including its architecture, knowledge model and some efficiency measures. Section three discusses related research efforts. Section four introduces the EECA model and finally section five concludes the paper.

2. REFLEX Overview

The work in this paper builds on the active database extension, to existing object-oriented databases, embodied in the REFLEX model. REFLEX is an active database prototype that is *loosely-coupled* to the underlying database management system. It has been designed to allow the *'bolting-on'* of the REFLEX extension to an existing object-oriented database system. Thus providing active facilities to an organisation's existing database. This has a number of advantages, firstly it makes active databases available today. Secondly, and perhaps more importantly, it allows an organisation to protect its investment in technology, resources and legacy systems. Active functionality by means of augmenting an existing and proven DBMS provides security and a migration path to the user community. Using REFLEX, new applications can utilise active functionality, whilst existing non-active applications can continue to use the host DBMS as normal. This option is available for applications that do not perceive a requirement for active functionality currently, but allows for the provision if the functionality is required at a later date, see figure 1. A brief summary of REFLEX will be given here but for further details refer to [NI93a, NI93b].

2.1. Architecture

As an active database system, REFLEX has to manage knowledge as well as data. To facilitate this, REFLEX has a Knowledge Management Kernel (KMK) which is central to its architecture. The KMKs major task is that it acts (a) as a command dispatcher to other constituent modules, and (b) as a rule evaluation scheduler. The other major modules of REFLEX are the Transparent Interface Manager (TIM),

the Event Manager (EM), the Knowledge Selection Module (KSM), the Condition Evaluation Module (CEM) and the Execution Supervisor (ES), see figure 2. The adaptive and loose-coupling features are afforded by TIM, which accomplishes most of this task since it allows the monitoring of *internal* events, and flags them to the EM.

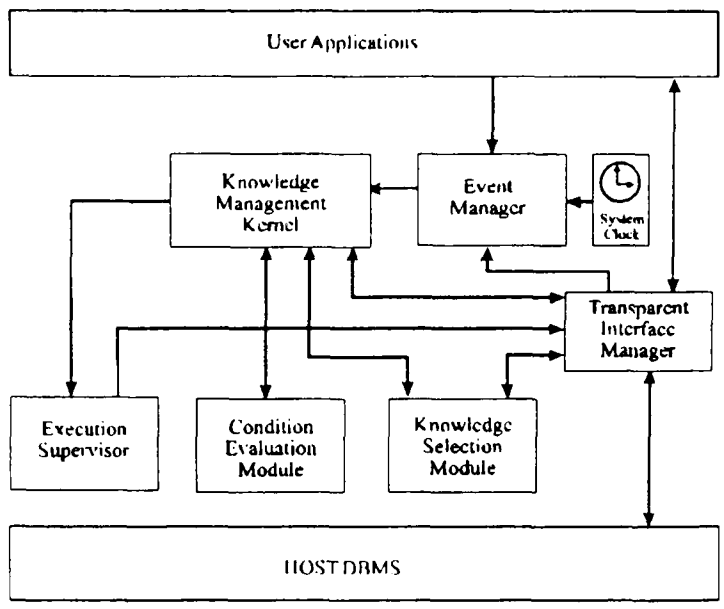


Figure 2 REFLEX Architecture

In REFLEX, it is possible to specify simple or composite events. Unlike a simple or primitive event which is regarded as being instantaneous, a complex event is composed over an interval of time (the interval being the duration from the occurrence of the first valid event to that of the final event in the specification). However, a complex event is said to occur at the point of occurrence of the final event in the specification. Hence complex events have chronologies or histories. These chronologies are realized by the EM time-stamping each occurrence of an event, before it informs the KMK of their occurrence, see section 4 for further details covering the semantics of the knowledge model.

When an event occurs, KMK instantiates a copy of the KSM (currently a uni-processor reentrant module), in order to evaluate if the event specification clause of any one or more of the rules has been satisfied and the rule has attained a state of '*in-context*'. If the event specification clauses are simple or primitive, then they are returned immediately to the KMK with the state of *in-context* set. If, however, the specification is complex, the KSM checks the *temporal log* to test if any relevant events have occurred previously. If so and the event occurred within a valid interval, the specification is again tested. If it succeeds, the KMK is informed as described earlier. Otherwise, the rule's part-satisfied event clause, is written to the *pending log* and kept for a given period of time, until its event clause is either satisfied or discharged on becoming invalid. The log maintains a copy of the state of rule evaluation ready for further events.

If a rule is in-context, it is passed to the CEM by KMK, to test its condition predicate. A condition can be declared in one of four ways: i) REFLEX's high level Object SQL dialect, ii) the proprietary language of the host database, iii) by *calling* an external module or iv) by having a NULL condition, i.e. it is always TRUE. If the condition clause is satisfied then the action clauses are executed subject to the condition-action (CA) coupling modes.

2.2. Knowledge Model

Rules are modelled as *first-class* objects, as in HiPAC [CBB*89] and ADAM [DP91, DPG91]. This is so rules to be handled in the same homogenous manner as the other objects in the database. Hence maintenance is simpler since the underlying DBMS maintains the rules as well as the data.

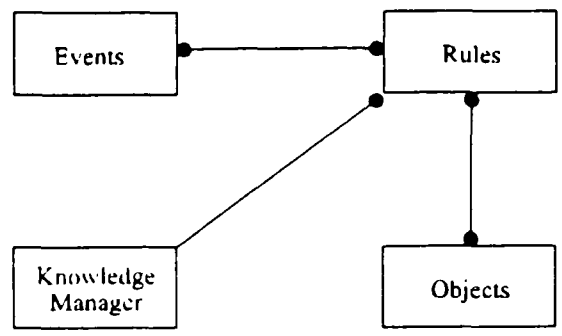


Figure 6 Knowledge Model

An object may have many rules applicable to it. Rules can be assigned to classes or to individual instances of objects. The object instances can also hold *exemptions* from certain rules as required. For each active database system, there must be one and only one Knowledge Manager. The Knowledge Manager is also modelled as an object. There may be many active database systems running against any one host DBMS. The relationships between the entities is illustrated in figure 3.

2.2.1. Event Objects

Most other active prototype systems, e.g. HiPAC and O_2 [MP90], model events as attributes. This provides fast execution and interpretation of events but is an inflexible approach. What can a user or developer do once a new event is to be added? Can the new event be added? If so, how? The answer is it can but with great difficulty and certainly not cheaply. To add the new event, the underlying active database system code must be modified, and recompiled by an active database system programmer. These modifications are costly in both monetary and system time dimensions. It may be infeasible to recompile a live database management system.

REFLEX models *events* as *first-class* objects. This provides a flexible approach, as the developer may add events at will. On first analysis, it may seem that having events as first-class objects, may cause severe degradation of service. This is because, on an event being raised, the event object must first be retrieved from the database, before its raise method can be called. Inherently, it seems to be plagued with intolerable overheads (seek, retrieve, call etc), on the other hand if events were treated as attributes, the knowledge base would have to be searched to determine those rules that are in context. This overhead can be countered by the utilization of the event object. As an object, the event has access to standard object modelling techniques. The most important being the complex object facility. Each event can maintain a list of rules to which it may apply. On the raising of any event, the KSM has immediate access to the rules which are brought into context by the particular event. Hence, the system is much faster at searching through its knowledge base, on an occurrence of an event. This feature becomes much more evident as the size of the knowledge base grows and can be aided by frequently occurring events being held permenantly in fast memory.

3. Related Research

Intensive research into active databases is being undertaken by many research centres. The widely known of these active databases are HiPAC [CBB*89], POSTGRES [SK91, SHP89], StarBurst [LLP*91], Ode [AG89] and ADAM [Pat89, DPG91].

REFLEX is, unlike many other ADBMS research prototypes, *loosely-coupled* to its underlying database. It is implemented upon commercially available DBMSs. It is designed to be portable and adaptive to many host DBMSs. Other research prototypes have not been afforded this flexibility as their active features are built into the database.

Most of the active database research prototypes use the Event-Condition-Action (ECA) model reported in McCarthy and Dayal [MD89] during their work on the HiPAC project. This ECA model is now a dominant and almost exclusive knowledge model used within the active database community.

Gehani et al [GJS92a], have proposed a new Event-Action (EA) model which folds the condition part into the event specification. The main reason given was that with the ECA model, there were simply too many coupling modes for the active database to deal with. The attraction of the EA model is that it does indeed do away with the many coupling modes. It does however, limit the functionality of the overall system for a number of reasons. The first and most obvious disadvantage is that in order to test the event clause, which includes the condition statement (including any required mask), the evaluation of the clause is sought with undue inefficiency. This is caused, we understand, by the evaluation of conditional statements even if they were not brought into context by the triggering event i.e. the event specification alone was not satisfied. The result of the event clause is not known until the conditional part of the specification was also tested. The second disadvantage is more subtle and is not handled by any other active models, since they do not handle external conditions. If the condition part of the clause is based on the state of the external environment, rather than that of the internal environment of the database, this cannot be extracted from the integrated event and condition clause that Ode proposes. For Ode or other current active databases to handle the condition based on the external environment, dummy updates are required to the database in order for the internal condition evaluation to take place.

The REFLEX model addresses this scenario, see section 4.1 on the scope of condition clause.

4. The Extended Knowledge Model

The REFLEX prototype has allowed the investigation of real data use in an active database system, using live applications. These investigations have highlighted several omissions of the standard ECA model, for example the replication of rules, and the creation of negative rules. This section examines some extensions to the ECA model, provided by REFLEX. The temporal algebraic event specification language is also

described.

4.1. Scope of the Condition Clause

Most of the current active database prototypes, as far as we are aware, allow the condition clause to be declared using some sort of Data Manipulation Language (DML) query. We recognise that this form of condition declaration is useful, as it allows the user or designer to use a familiar interaction protocol. However, it is also limiting as it forces the designer to initiate unnecessary access to the database, thus adversely affecting the performance of the overall system.

Since an active database responds to changes in its environment, the above form of the condition clause addresses *only one* aspect of the total environment, that is the internal state of the database. REFLEX however, allows the *calling* of user defined condition modules. This provides support for changes in the environment which require a complex condition statement which cannot be handled by the DML language. Or the condition requires access to external or application specific parameters, possibly user initiation, which have no bearing onto the internal state of the database. The external condition module simply returns a boolean of TRUE if the condition statement is satisfied or FALSE otherwise.

This extension allows all the sections of the EECA tuple to independently access either internal or external states of the environment.

4.2. Situation Redundancy

There may be situations (both events and conditions) common to many rules, but each with alternate actions i.e. the same situation in the environment triggers these rules.

If events are raised which bring into context many rules, the event specification clauses of these rules must be evaluated. After the event specification clause has been evaluated, the condition clause must also be evaluated. If the situation of the rules, are the same, then there has been multiple or redundant evaluation of many rules event and condition clauses. Causing the overall system to be inefficient.

The proposed EECA model alleviates the problems associated with redundant situation declaration by allowing a rule to have multiple actions. This also implies that a rule must have multiple Condition-Action coupling modes.

There are occasions where it is easier to state a negative condition rather than a normal condition, as it may be far more efficient to evaluate. The EECA model accommodates this situation by using a construct that is similar to an *else* statement in conventional block structured programming languages. For this case the EECA model proposes *Fail Actions*. These are actions that may be executed if the condition clause of the

rule fails (or does not hold). Multiple fail action clauses are also permitted within the EECA model, along with their respective Condition-Fail-Action coupling modes.

A rule in the REFLEX Knowledge Model is represented as:

ON	event specification
IF	condition holds i) internal: NULL, OSQL, or prop. language ii) external
THEN	execute action 1
	...
	execute action n
ELSE	execute fail action 1
	...
	execute fail action n

The Action and Fail-Action clauses are mutually exclusive, just as with the THEN-ELSE structure. The clauses may contain requests to abort the parent transaction, undertake some DML query or call some external module.

4.3. EECA Coupling Modes

Coupling modes couple one part of the ECA triple to its subsequent neighbour. The three common coupling modes are:

i) immediate, where the parent transaction is suspended while the child transaction is being executed.

ii) deferred, where the child transaction is deferred until the parent transaction is completed, at which point the child is executed

iii) decoupled or separate, where the child transaction is processed concurrently with the parent transaction.

To these coupling modes, the complex issues of dependence need to be addressed. Is the committal of the child process dependent on that of its parent? The reverse is more serious. Is the committal of the parent dependent on that of its child? What would be the outcome where in the simplest case of an immediate coupling mode, the parent is ready to commit but the child aborts, will the parent commit or abort? A more interesting problem is posed if the same question were raised but for a decoupled coupling mode. Is the committal of the parent dependent on that of a decoupled child?

The EECA model requires that all the action statements (including fail actions) for each of the rules have a flag that signifies whether the action is dependent or independent with respect to its initiating transaction.

The onus for dependence has been passed to the designer of the system. Hence the action clause is effectively an object or tuple (with arity 3), as is demonstrated below:

Action clause	(execute action 1, coupling mode, dependency flag)
	...
	(execute action n, coupling mode, dependency flag)
Fail Action clause	(execute fail action 1, coupling mode, dependency flag)
	...
	(execute fail action n, coupling mode, dependency flag)

This then leads to the semantics of the coupling modes. The reader may note that for the condition clause the available EC coupling modes are unchanged i.e. the condition clause can have one of the following coupling modes: immediate, deferred or decoupled. All three modes are offered the option to be dependent or independent of the parent transaction. If, for a given situation, where there are many actions, the declaration above of using multiple actions clauses, may only be used if the EC coupling modes are the same. If the EC coupling modes are different, then different rules need be declared. This design decision was taken so that the rule declaration was not over complicated with many excess coupling modes for situations which would hardly arise.

4.4. Event Specification Language

REFLEX supports the notions of complex (composite) events, as do other active systems such as HiPAC and Ode. Simple or primitive events are relatively easy to understand. They are said to occur at a specific point in time, unlike conditions which *hold* over certain intervals or periods of time. Complex events blur this distinction. A complex event is said to occur at the point at which the last valid component (primitive) event occurred. The component events have the property of validity. An event is only valid for a given interval, after which it is no longer valid for a certain event specification. The occurrence of the event may still be valid for a different rule's event specification. Events can be internal to the database (DML commands e.g. updates, reads and transaction points), temporal (at specific points in time, relative or periodic) or the events may be abstract i.e. externally defined by user applications. All non-temporal events have an interval of occurrence i.e. **BEFORE** or **AFTER** an event point.

The temporal event algebra used by REFLEX is very powerful. Even so, ease of use has not been compromised as standard English statements are used to declare the powerful clauses. The algebra contains several logical and temporal keywords. The logical keywords are **AND** (unordered conjunction of $E_1 \wedge E_2$), **OR** (inclusive disjunction of $E_1 \vee E_2$), **XOR** (exclusive disjunction of $E_1 \veebar E_2$), **NOT** (negation $\neg E$), **PRECEDES** (sequenced conjunction of $E_1 \cdot E_2$) and **SUCCEEDS** (sequenced conjunction of $E_1 \cdot E_2$). The temporal keywords are **BEFORE, AFTER, AT, BETWEEN, ON, WITHIN HOUR/MIN/SEC, EVERY HOUR/MIN/SEC, MIN, MAX, DATE, TIME.** The **EVENT** keyword precedes abstract or user-defined (external) events. The negation operator may

not reference temporal events e.g. NOT 5:00pm, as this would cause the event to be raised every millisecond (or machine clock granularity) that was not 5:00pm. Parenthesis are used to override operator precedence.

Examples of the english event specification language (ESL) are:

read student	simple internal event - read
before update account **or after** update employee	$\vee$ - internal events
Event$_1$ **precedes** Event$_2$ **within hour** 24	$E_1 \triangleright E_2 \wedge ((t_{E2}-t_{E1}) < 24$ hour)
at 5:00pm **every** friday	periodic
event radar	user-defined or abstract

The rule may reference the object that raised the event by referencing the position in the event specification clause, and by using the OBJECT keyword. For example, in the above *read student* example, if the condition clause wanted to reference the raising object it would use OBJECT1 as the student class is the first mentioned class (it is the only class in this example). Similarly to reference employee in the second example, OBJECT2 would be used.

Example EECA rules could be:

E	AFTER UPDATE aircraft	AFTER UPDATE item
C	SELECT a.Name() FROM aircraft a, aircraft b WHERE a.Name() = OBJECT1 AND (a.CurX - b.CurX) BETWEEN -5 AND 5 AND (a.CurY- b.CurY) BETWEEN -5 AND 5 AND (a.CurZ - b.CurZ) BETWEEN -5 AND 5;	SELECT a.Name FROM item a WHERE a.Name = OBJECT1 AND a.QtyOnHand < a.MinQty;
EC	immediate	deferred
A	(AlertOperator OBJECT1; immediate; independent) (INSERT ON log a.itemID, XYZ; decoupled; independent)	(INSERT ON reorderItem a.itemID, a.ReorderQty; decoupled; independent) NULL
FA	NULL	NULL

5. Conclusions and Future Work

We have introduced the EECA model, with its multiple action and fail-action clauses, and its associated extension of coupling modes. This paper has highlighted several problems of application semantics caused by the EECA polyform, mainly the dependency issue. This has been resolved by introducing the action clause tuple that includes a dependency flag for each individual action or fail-action clause. The designer

of the system is given the choice as to what level of transaction dependency is required for a given application.

We believe that the EECA knowledge model proposed does in fact allow the declaration of the knowledge within the active database system to be both semantically concise and obvious as to its intention. The model also allows for a more efficient evaluation and operation of the overall active database system.

REFLEX introduces a number of novel features such as its loose coupling model, its powerful knowledge model, its self-activity and its short-circuit evaluation mechanism. REFLEX also promotes a critical concurrency approach, the concept of non-destructive knowledge, a powerful and user-friendly graphical user-interface (Vis), see [NI93c] for further details.

The REFLEX prototype system has been implemented in C++ on the ONTOS ODBMS [ONT91]. It *has been demonstrated* at various venues using a graphical simulation of an Air Traffic Control System. The prototype is currently being used to generate data on how *real active applications behave*.

We intend to make REFLEX available on the public-domain, via ftp. It will initially be released for the ONTOS DBMS system. Please contact the authors for further details.

References

[AG89] Agrawal R. and Gehani N.H., "Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++", 2nd Int. Workshop on Database Programming Languages, Portland, OR, June 1989

[CBB*89] Chakravarthy S., Blaustein B., Buchmann A. et al, "HiPAC: A Research Project in Active, Time-Constrained Database Management", Final Technical Report, Xerox Advanced Information Technology Division, July 1989

[Day89] Dayal U., "Active Database Management Systems", Sigmod Record, Vol. 18, No. 3, 1989

[DP91] Diaz O. and Paton N.W., "Sharing behaviour in an object-oriented database using a rule-based mechanism", Proc. 9th British National Conference On Databases, 1991

[DPG91] Diaz O., Paton N. and Gray P., "A Rule Management in Object Oriented Databases: A Uniform Approach", Proc. of the 17th Int. Conf. on Very Large Data Bases, Barcelona, Spain 1991

[GJ91] Gehani N.H. and Jagadish H.V., "Ode as an Active Database: Constraints and Triggers", Proc. of the 17th Int. Conf. on Very Large Data Bases, Barcelona, Spain 1991

[GJS92a] Gehani N.H., Jagadish H.V. and Shmueli O., "Event Specification in an Active Object-Oriented Database", Proc. 1992 ACM SIGMOD Intl. Conf. on Management of Data

[GJS92b] Gehani N.H., Jagadish H.V. and Shmueli O., "Composite Event Specification in Active Databases: Model & Implementation", Proceedings of the 18th Int. Conf. on Very Large Data Bases, Vancouver, Canada, 1992

[LLP*91] Lohman G. M., Lindsay B., Pirahesh H. and Schiefer K. B., "Extensions To STARBURST: Objects, Types, Functions, and Rules", CACM October 1991, Vol 34, No 10

[MD89] McCarthy D.R. and Dayal U.,"The Architecture of an Active Data Base Management System", Proc. ACM SIGMOD Intl. Conf. on Management of Data, Portland, June 1989

[MP90] Medeiros C.B. and Pfeffer P., "A Mechanism for Managing Rules in an Object-oriented Database", Altair Technical

Report, 1990

[NHI94] Naqvi W., Hughes C., and Ibrahim M.T., "Towards a Dynamic Schema Integration Model", Tech Report CIT-DSRL029401, University of Greenwich, December 1993, submitted for pubication

[NI93a] Naqvi W. and Ibrahim M.T., "REFLEX Active Database Model: Application of Petri-Nets", Proc. of the 4th Int. Conf. on Database and Expert Systems Applications, Prague, September 1993

[NI93b] Naqvi W. and Ibrahim M.T., "Rule and Knowledge Management in an Active Database System", Proc. of 1st Int. Workshop. on Rules in Database Systems, Edinburgh, September 1993

[NI93c] Naqvi W. and Ibrahim M.T., "The REFLEX Knowledge Acquistion User Interface", Tech Report CIT-DSRL12932, University of Greenwich, December 1993

[ONT91] "ONTOS Reference Manual", ONTOS Inc, 1991

[Pat89] Paton N.W., "ADAM: An Object-Oriented Database System Implemented In Prolog", Proc. 7th British National Conference On Databases, 1989

[SHP89] Stonebraker M., Hearst M. and Potamianos S., "A Commentary on the POSTGRES Rules System", Sigmod Record, Vol. 18, No. 3, September 1989

[SK91] Stonebraker M. and Kennitz G., "The POSTGRES Next-Generation Database Management System", CACM October 1991, Vol 34, No 10

REFLEX Active Database Model: Application of Petri-Nets

Waseem Naqvi and Mohamed T. Ibrahim

Database Systems Research Laboratory
University of Greenwich, London, SE18 6PF, U.K.
{w.naqvi, m.ibrahim}@greenwich.ac.uk

Abstract. REFLEX is an active database research prototype, designed to provide a flexible and adaptive active database extension to an existing database system. This paper reviews the REFLEX model and its architecture. Some of the main contributions of the research are discussed, such as the notion of *self-activity* and of enabling investments in legacy systems to be preserved. Some current applications of petri-nets to rule management are described before the design and modelling of REFLEX using petri-nets.

1. Introduction

Active databases have generated substantive interest from the research community. In essence these systems encapsulate an enterprise's domain knowledge within the system. Generally most active prototype systems have the notions of event-condition-action (ECA) [C1]. REFLEX, is a research prototype of an active database system designed for implementation as an extension for existing organisational databases. It builds on and extends notions and concepts from other related research prototypes along a number of dimensions. The status of these extensions in the current prototype and those planned for future work are described.

The philosophy of the REFLEX architecture and design encompasses the provision of a flexible, adaptive and *active* capability to an organisation's existing database. The benefit of such an approach, in addition to the aforementioned features, is that it preserves an organisation's investment in legacy systems, resources and training.

Petri-nets [P2] have been proven as viable modelling and analysis techniques. This paper reports on the current prototype's design philosophy and features as well as the current status of modelling and analysing the REFLEX system using the Petri-net theory.

The paper is organised as follows, section 2 reviews the REFLEX model and its architecture. Section 3 briefly surveys expert and knowledge based systems. Following on, section 4 describes the application of petri-nets to the formalism of the REFLEX model. Finally section 5 concludes with future directions.

2. The REFLEX Model

An active database management system (DBMS) must be able to manage knowledge as well as other notions of its activity feature. Thus, the architecture of an active database management system, in contrast

to that of a passive DBMS, must deal with *rules*. It may also support other types of knowledge representation schemes e.g. frames in order to enhance the functionality of the system as explained in later sections. Rules would then become part of the frame representation scheme. Various active monitors or daemons could then be attached to one or more of the rule's objects: events, and/or conditions and/or actions. The main types of the triggering events are given in section 2.3.

2.1. REFLEX Architecture

The main components are: the transparent interface manager, the knowledge, event and transaction/execution models. Three of these components are common to many other active prototypes. REFLEX, however, differs from other prototypes in that it provides a Transparent Interface Manager (TIM), Fig. 1, to a host DBMS. Because of this transparency, TIM contributes to the flexibility and adaptability features of REFLEX as mentioned above. Further details about TIM are explained in section 2.2.

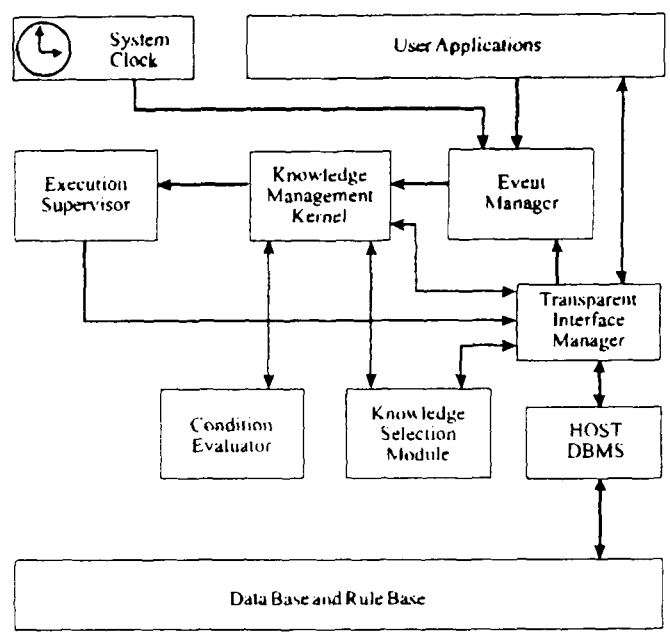


Figure 1. REFLEX Architecture

The REFLEX knowledge model, as mentioned above, combines two schemes of knowledge representation, namely production-rules and frames [R1]. REFLEX is thus able to support the cause/effect and deep knowledge reasoning by the provision of these two types of knowledge representation .

The rule management system implements a priority mechanism. During a critical period only the highest priority rules will either be executable or can preempt other lower priority rules. The rule priority system is also used in rule conflict resolution.

Unlike POSTGRES [S1, C2], REFLEX is designed and built as an object-oriented system. It also differs from other prototypes in one important aspect. Whereas other research prototypes, such as HiPAC [C1] and POSTGRES, are tightly coupled to their underlying model, REFLEX implements a *loose coupling*. This feature is used to provide *'activity'* for many models i.e. object model, relational model and the extended relational model. A comparison of the features of a number of active database prototypes, which were not included because of space limitation, are given in [N1].

The *'active'* notion plays a dual role in our prototype. The first role, common with other prototypes, is *imparting* active capability into the application domain. The second role which, as far as we know is one contribution of this research, is that REFLEX employs a ***'self-active'*** capability to update itself. The knowledge base (KB) as well as an application database are stored within the REFLEX system. Thus the maintenance of the KB can also be subject to the notion of activity. As an example, the rule's state is monitored *actively* by the REFLEX system. Rules have three components: events, conditions and actions. The clauses for each of these components are compiled or recompiled at the point of rule creation or on

rule modification. The re-compilation process being automatically triggered on a rule change. Note that the issue of knowledge base validation plays a crucial part in this case, see section 6. It will be assumed for the moment, however, that the knowledge base is consistent, valid and complete in the sense of [J4].

Since one of the main target uses of REFLEX is real-time/safety-critical systems, the REFLEX system must function in an efficient and safe manner. It must also ensure that the design of applications running under REFLEX is formally validated and verified. Petri net technology and formalism is used in the analysis of REFLEX's design and construction. A suite of tools are also being designed to support the development process.

2.2. Outline of the Model

Transparent Interface Manager (TIM). TIM allows *internal* database events to be monitored, and highlights the events to the Event Manager. Any access to the database must go through TIM. The underlying database's transaction manager has been harnessed, using *wrapper* technology, to allow the detection of transaction events and also to allow the creation of nested and sibling transactions. The transaction manager interface has special wrapper functions which trap all calls to the transaction manager, as the code wedges in symbol tables, and informs the Event Manager that a transaction based event has taken place.

Event Manager (EM). Other research prototypes, e.g. HiPAC [C1] employ the notion of primitive and composite events and both an event and '*history*' algebra in their event model. REFLEX adopts these concepts from the HiPAC model. In the following section, we describe the working of this module.

When an event is detected, the Event Manager logs it in the temporal log. Further processing steps depend on whether the event is simple or complex and will be explained in section 2.3. EM then informs the Knowledge Management Kernel that an event has occurred.

Knowledge Management Kernel (KMK). The KMK acts as the nucleus of the REFLEX architecture. KMK's major tasks are (a) as a command dispatcher to the other modules, and more importantly, (b) as a rule evaluation scheduler. REFLEX is designed to be used in many areas and particularly in the real-time application domain. Hence, it must respond very quickly to be able to influence the environment of which it is a part; e.g. in process control applications. For this reason many of the modules are designed to operate concurrently. The KMK is responsible for scheduling the cooperating modules which frequently act on the same data space. The interfaces between the KMK and the five modules are described below.

a) EM-KMK-KSM Interfaces. When the EM informs the KMK that an event occurred, KMK evaluates whether it affects any rules. The list of affected rules is passed to the Knowledge Selection Module (KSM) which returns a list of '*in-context*' rules i.e. their '*ON*'-clause are satisfied. These rules are passed to the Condition Evaluation Module, subject to the event-condition coupling mode.

b) KMK-CEM-ES Interfaces. Depending on a rule's priority level and its coupling mode, KMK simultaneously dispatches it to the Condition Evaluation (CEM) and the KSM modules. This design strategy ensures parallelism and faster availability of the two results. If both results are true, the rule's action clause may be passed to the Execution Supervisor immediately. However, if the KSM returns an unsatisfiable rule, the CEM is preempted, and the result discarded. If the CEM returns its result first, it can be written onto the temporal log ready for use.

On return from the CEM, if the rule's condition clause has been satisfied (rule is *fireable*), it is then passed to the Execution Supervisor, subject to the condition-action coupling modes. Otherwise the rule is discarded from working memory and no further action need be taken.

Knowledge Selection Module. On being called by KMK, the KSM recalls the set of rules that the event affects. If a rule has a simple event of the kind that occurred, it is returned to the KMK with a status of *'Yes in context'*. If the rule specifies complex events, the KSM checks the temporal log for any relevant related events that occurred previously. If so, it checks if the event specification is satisfied, and informs the KMK. If a rule's event clause is part-satisfied, it is written to the *pending* log and kept for a given period of time, until its event clause is either satisfied or discharged. The log keeps a copy of the state of rule evaluation, ready for further events to be raised.

Condition Evaluation Module. The rule's condition predicate is evaluated. A condition can be in: (a) REFLEX's high level Object SQL dialect, (b) proprietary language of the host DBMS. REFLEX maps the Object SQL to the proprietary language. An application designer thus has the flexibility to write the clause in either form. The rule's condition clause is compiled, as with the other clauses, either at creation time or on modification. Examples of some rule condition clauses follow:

```
ON      update customer                    ON      update stock
IF      customer.name = 'Fred Bloggs'          IF      item.qty < = item.reorder_qty
        AND customer.credit_limit = 0          THEN    insert onorder item, reorder_qty
THEN    call Alert_No_Credit
```

Execution Supervisor. The rule's action clause is evaluated. The clause can be in one of two dialects: (i) a query in object SQL or (ii) a call to an application defined object (process module).

2.3. Event Specification

The event clause is expressed using algebra which expresses temporal relationships between events. Event histories are maintained in the temporal log to support the evaluation of complex event clauses. Our time model is based on *interval logic*, with all events having a before or after (default) granularity.

The types of events are: (i) Internal, Object events: before/after create, get, etc. Transaction events: before/after start, commit, abort (ii) Temporal events: at (specific-date/time), periodic, after (duration), sequential (iii) External events: are user-defined. Events are augmented with the concept of *validity*

whereby a valid event occurs within a certain period. Example event specifications and rule syntax are:

a) Event$_1$ **AND** Event$_2$ **WITHIN MIN** 30

Event$_1$ **PRECEDES** Event$_2$ **WITHIN HOUR** 24

b) ON event-algebra

IF either i) no condition ii) Object SQL query iii) proprietary language

THEN either i) call user module ii) Object SQL query iii) proprietary language

i) event algebra clause.

update <table/class/object name>

on an update or modification of a table,class, or instance object

update <table/class/object name> AND TIME 5.00pm

ii) condition clause. The clause may be set to TRUE (i.e. no condition), or expressed in either Object SQL or the host DBMS's proprietary language.

e.g: *TRUE* i.e. no condition, just an Event-Action pair.

or *class_name.attribute* {=, <, <=, >, >=, !=} *expression*

iii) action clause. The clause may call an object module or as, with the condition clause, a more complete query may be expressed in either Object SQL or the host DBMS's proprietary language.

e.g. *call* program module (object) or *delete* object

2.4. Current Status

Phase one of the model has been implemented using AT&T C++ (v2.1) and the ONTOS object-oriented database management system. The user interface was built upon the X11R5 windowing system using a Sun Sparc Workstation running Solaris 1.1. Another implementation of the REFLEX prototype is being mirrored using the P.O.E.T. database and MS Windows on a PC platform.

3. Applications of Petri-Nets

Petri-nets and knowledge based systems have been used to help solve problems in many application areas. Applications include manufacturing, process control, planning, decision support, medical applications, etc. [L1, A1, B2, S1].

Petri-net (PN) models are abstract and formal representations of information flow [P1]. Their major use has been in the definition, design, modelling of systems that exhibit concurrent behaviour using their well known powerful modelling and analysis properties.

In their paper [B3], Bemaiza et al describe a knowledge-based Petri-net experimental tool for modelling and analysis which consists of a graphical editor to capture and edit Petri-net graphs. Their expert system is based on rules capturing the general knowledge about reachability and invariance of Petri nets. Their approach was investigated as a possible candidate for use in the analysis of REFLEX.

However, whereas [B3] uses Prolog, an object oriented approach using C++ and a knowledge-based system (e.g. KAPPA) is under study.

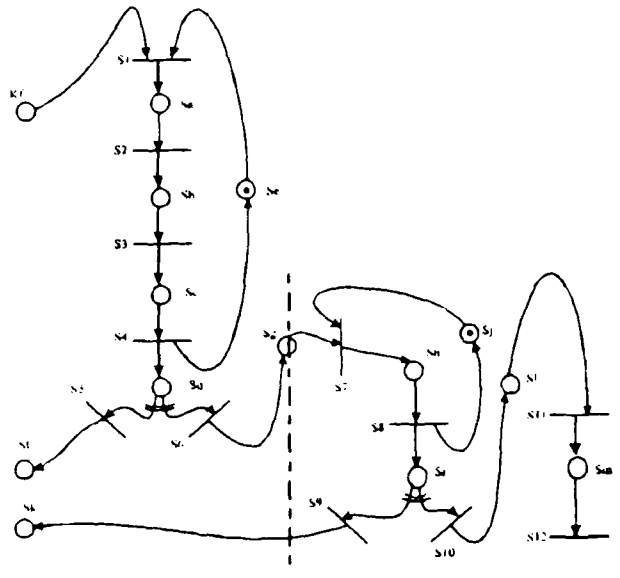


Figure 2 Petri-net Graph of Knowledge Selection Module

4. Petri-nets in the REFLEX model

We share in the belief's of many others [R2, R3] that petri-nets are powerful techniques for the design and analysis of many systems. When coupled with object technology and knowledge-based systems they provide a powerful environment for many domains including active object oriented database systems. We are not aware of any research projects reported where these three technologies are being investigated and integrated in the area of active and/or object-oriented systems. Petri-nets are an instrumental technique in providing formal definitions and a sound basis for validating the design of REFLEX.

It is well known that production rules are the most widely used knowledge representation scheme. These systems, despite their many desirable features e.g. principally their simplicity, lack the power to '*model* ' the behaviour of many real world systems. REFLEX aims to combine rule-based and frame-based knowledge representation schemes in order to provide cause/effect deep knowledge and reasoning mechanisms.

The notions of event-condition-action (ECA) is used in REFLEX, and other research prototypes [C1], together with the underlying architecture that includes a transaction execution model; where the coupling between groups of ECA and their execution with respect to their triggering transaction, this feature distinguishes active databases from KBs. [C1] suggest that requirements may vary with the application. Real-time applications might require an *immediate* (coupling) mode of execution to evaluate a condition as soon after an as event occurs and without waiting for the whole of the triggering transaction to complete. REFLEX has been designed in discrete units. Since it is envisioned that the final prototype will be implemented on multiprocessor hardware.

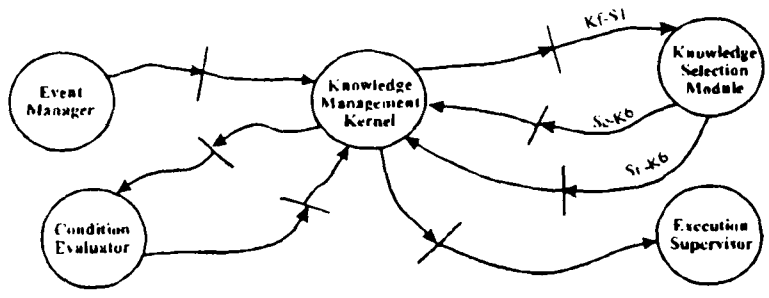


Figure 3 REFLEX Context Petri-net

For illustration the KSM is represented as a petri-net, Fig. 2. KSM was formally verified using the techniques of reachability analysis, and safeness checking.

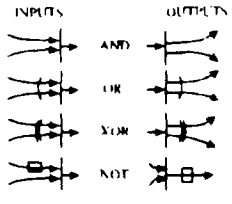


Figure 4 Petri-net Logical Notation

A high level context petri-net graph of the REFLEX architecture can be found in Fig 3. For REFLEX we have adopted a similar approach to [B1] and augmented the Petri-net theory by adding the following enhancement

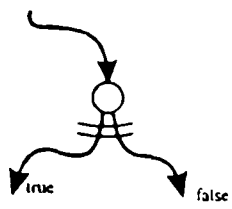


Figure 5 if-then construct

(Fig. 4).The above can be used singularly or combined to perform additional logic. To provide an if-then or case construct, Fig. 5, is used in this paper.

A Petri-net structure, C, for the Knowledge Selection Module follows, Table 1.

From the initial markings μ = (1,0,0,0,0,1,0,0,0,0,1,0,0,0), the reachability tree is produced, see Fig 6. The reachability tree is the standard analysis technique for petri-nets, which verifies that all nodes can be reached and that infinite loops do not occur.

The petri-net graph for structure C can be found in Fig. 2. It can be observed that the KSM can be split into two distinct modules. On the left side, the basic event-rule matching and event specification. The right side, to action and satisfy the complex event specification of the rules in context. These two modules can execute concurrently. The two modules exhibit the familiar *producer-consumer* scenario with the place **Sg**, Fig. 2, representing the buffer or queue between the two.

C = {P,T,I,O}

P = { Kf, Sa, Sb, Sc, Sd, Se, Sf, Sg, Sh, Si, Sj, Sk, Sl, Sm}

T = { S1, S2, S1, S4, S5, S6, S7, S8, S9, S10, S11, S12}

I(S1) = {Kf} O(S1) = {Sa}
I(S2) = {Sa} O(S2) = {Sb}
I(S1) = {Sb} O(S1) = {Sc}
I(S4) = {Sc} O(S4) = {Sd, Se}
I(S5) = {Sd} O(S5) = {Sf}
I(S6) = {Sd} O(S6) = {Sg}
I(S7) = {Sg} O(S7) = {Sh}
I(S8) = {Sh} O(S8) = {Si, Sj}
I(S9) = {Si} O(S9) = {Sk}
I(S10) = {Si} O(S10) = {Sl}
I(S11) = {Sl} O(S11) = {Sm}
I(S12) = {Sm}

Table 1 Petri-net Structure

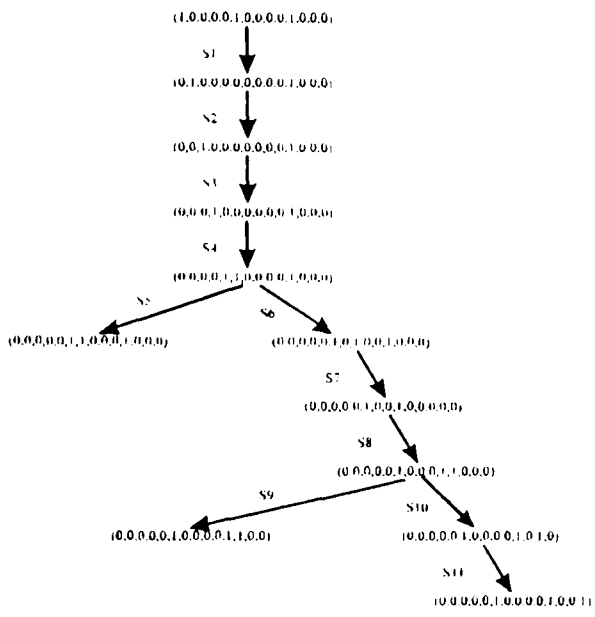


Figure 6 Reachability tree for Knowledge Selection Module

Analysis of Petri-Net Graphs

A complete analysis of petri-net graphs requires detailed study of many issues. This is especially true for a large system such as REFLEX. It must be shown that all places can be reached, that the system is safe and within specified bounds. Petri-nets offer many analysis techniques such as: safeness, boundedness, conservation, reachability and coverability [P2].

5. Conclusions and future work

This research aims to establish the feasibility of the loose-coupling approach between REFLEX and the underlying data model of a host database. This is easily achievable if REFLEX is implemented on parallel, distributed architecture. However its viability on a single-processor model, remains to be established. However, with the rapid increase in client/server popularity, it is possible for organisations to use REFLEX. As stated earlier, this will enable these organisations to preserve their investments in legacy systems and training. This is one contribution of this research in the area of active database research. Another contribution is the integration of Petri-nets, object-orientation and active database technologies.

As to future work, there are many issues still to be resolved. One important issue is the need to check

that the knowledge base is valid, consistent, complete and accurately represents a problem domain.

Jafar and Bahil [J4] describe a system for interactive verification of knowledge-based systems. However, we also believe that some aspects of the validation process can be beneficially automated. Following [O1], the questions that need to be answered include: what should be validated? how is it validated? what are the procedures for validation ? how is validation integrated into development?

Version management of the knowledge base for reasoning also needs to be considered if explanation of a systems behaviour over time is to be supported.

7. References

[A1] Agarwal R. and Tanniru M., "A Petri-net based approach for verifying the integrity of production systems", International Journal of Man-Machine Studies, Vol. 36 No 3 pp 447-468, March 1992

[B1] Baer J., Bovet D. and Estrin G., "Legality and Other Properties of Graph Models of Computations", Journal of the ACM, Vol. 17, No. 3., July 1970, pp. 543-554

[B2] Brenner E., Grabner J., Moosburger M., Otschko G., Schlögl K., Seifter P., Song J., Steger Ch. and Weiss R., "Design and Implementation of a Distributed Real-Time Expert-System for Fault Diagnosis in Modular Manufacturing Systems", Microprocessing & Microprogramming, Vol. 32 No 1-5 pp 799-806, August 1991

[B3] Benmaiza, M. and Elkaraksy, M.R., "Knowledge-based approach to Petri nets analysis", Knowledge-Based Systems Vol: 4 Iss: 3, pp. 144-56, Sept. 1991

[C1] Chakravarthy S., Blaustein B., et al, "HiPAC: A Research Project in Active, Time-Constrained Database Management", Final Technical Report, Xerox Advanced Information Technology Division, July 1989

[C2] Cattell, R.G.G, "Object Data Management: Object-Oriented and Extended Relational Database Systems", Addison-Wesley, 1991.

[J1] Jensen K., "Coloured Petri Nets: A High Level Language for System Design and Analysis", Advances in Petri Nets 1990, Springer-Verlag, Lecture Notes in Computer Science, 483

[J2] Jianjun Y., Feng Z., Jiati D. and Chuaijun C., "Intelligent Manufacturing cell controller IMCC-E", Human Aspects in Computer Integrated Manufacturing Conf., Tokyo, Japan, IFIP Transaction B, Vol. B-3 pp. 745-755, 1992

[J3] Jensen, K. (Ed.), Application and Theory of Petri Nets 1992, Proceedings of the 13th international Conference, Sheffield, UK, June 1992, Springer-Verlag, Berlin 1992.

[J4] Jafar, M., Bahill, T.A, "Interactive Verification of Knowledge-Based Systems", IEEE Expert, Vol.8 No.1, Feb. 1993.

[L1] Lipp H.P., "Application of timed fuzzy Petri nets in expert systems for operative management of complerx prodyction systems", Prozessreegensystemme '91 Conf. (Process Computer Systems '91), 1991, pp. 103-12 (in German)

[N1] Naqvi W. and Ibrahim M.T., "The REFLEX Active Database System", Database Systems Research Laboratory, University of Greenwich, Internal Report, 1992

[N2] Naqvi W. and Ibrahim M.T., "REFLEX: An Active Database Extension", BNCOD11, July, 1993

[O1] O'leary, T.J., Goul, M., Moffit, K.E., Essam Radwan, A., "Validating Expert Systems", IEEE Expert, Vol. 5 No3, June 1990.

[P1] Paterson J.L., "Petri Nets", ACM Computing Surveys, Vol. 9, No. 3, September 1977

[P2] Paterson J.L., "Petri Net Theory and the modeling of Systems", Prentice-Hall, 1981

[R1] Ringland G., "Structured Object Representation - Schemata and Frames", Approaches to Knowledge Representation, Ed. Ringland and Duce, 1987, pp 81-99

[S1] Stonebraker M. and Kemnitz G., "The POSTGRES Next-Generation Database Management System", CACM October 1991, Vol 34, No 10

Rule and Knowledge Management in an Active Database System

Waseem Naqvi

Mohamed T. Ibrahim

Database Systems Research Laboratory

University of Greenwich, London, SE18 6PF, U.K.

{w.naqvi, m.ibrahim}@greenwich.ac.uk

Abstract. Todays new applications require that reasonable inferences be made on the data within the database i.e. knowledge of the application domain is required. Knowledge is a higher level abstraction than the data or facts alone. Active databases strive to encapsulate an application domain"s knowledge within the database. REFLEX is an active database research prototype. It"s main tenet is that it provides knowledge management facilities for traditional existing database management systems. This short paper discusses the knowledge management facilities and the unique features that REFLEX provides such as its novel *concurrency mechanism*, *self-activity*, its *non-destructive knowledge* model, and its *graphical user-interface*.

Key words: active database, object-oriented, knowledge management, real-time systems

1. Introduction

There is a growing interest in moving knowledge from an application into a database or knowledge base. This has been attempted by knowledge bases, deductive databases and lately by active databases. In an active database the enterprise"s domain knowledge has been encapsulated within the system. The knowledge is centralized in one place, i.e. within the database management system itself, as opposed to being scattered across many application programs. Thus avoiding the problems this may cause, e.g. replication of knowledge, effort and possible inconsistencies.

Typically, most active prototype systems have the notions of event-condition-action (ECA) [2, 3] triples. They include three components: knowledge, event and transaction models. Even though some of these components are common with other types of systems e.g. knowledge and deductive database systems, there are important differences namely, the ability to encode richer and more diverse application logic which is triggerable automatically by the occurance of events in the database. The *coupling modes* between transactions also being major variation.

REFLEX, is a research prototype of an active database system. It is designed for implementation as an extension for existing organisational databases. It builds on and extends notions and concepts from other related research prototypes along a number of dimensions. The philosophy of the REFLEX architecture and design encompasses the provision of a flexible, adaptive and *active* capability to an organisation"s

existing database. The benefit of such an approach, in addition to the aforementioned features, is that it preserves an organisation"s investment in legacy systems, resources and training. REFLEX provides an easy to use, graphic driven, but very powerful active rule system, which may be augmented to an existing database.

The paper is organised as follows, section 2 examines the knowledge management scheme within REFLEX and includes sub-sections on how rules are added to the database, the distribution scheme, the self-activity features. Section 3 looks at related research. Finally section 4 concludes and highlights future directions. Candidate working prototypes of active applications are introduced in the appendices.

2. Knowledge in REFLEX

Active database systems must be able to manage knowledge as well as data. The REFLEX knowledge model combines two schemes of knowledge representation, namely *production-rules* and *frames* [11, 19]. Reflex is thus able to support the cause/effect and deep knowledge reasoning by the provision of these two types of knowledge representation.

REFLEX has been designed and implemented using object-oriented technology, to provide activity to traditional databases. REFLEX differs from other research prototypes [20, 1, 10, 6, 4], as it has a Transparent Interface Manager (TIM), a *gateway*. It is TIM that provides the flexible and adaptive features of REFLEX. A block diagram of the architecture can be found in figure 1. For a more indepth discussion of the model and architecture of REFLEX please see [15].

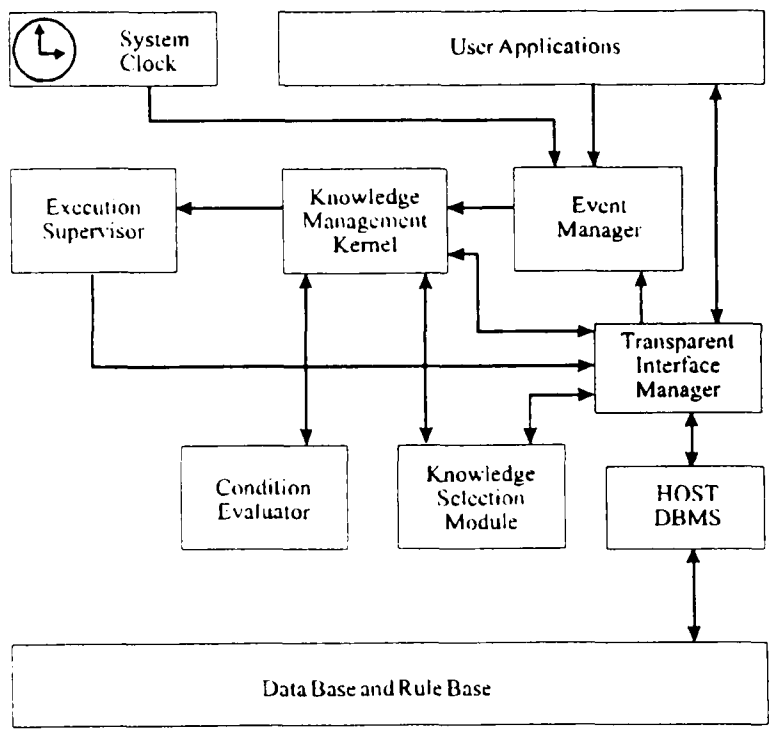


Figure 1 REFLEX Architecture

All access to the DBMS is routed through TIM. All existing applications work as normal, but if any applications require activity, TIM manages the activity. TIM allows internal events to be detected, such as database or transaction requests, mentioned later. TIM couples REFLEX to the underlying technology using "*wrapper*" technology, as host database calls are wrapped with a guard layer of code providing REFLEX information.

2.1. Structure of Knowledge

The structure of the rules in REFLEX have the following main attributes; Knowledge Management Kernel (the neucleus of the system), the list of objects a rule can act upon (class and instances), exempt

objects, list of applicable events, event algebra clause, condition clause, action clause, coupling modes (event-condition, condition-action), rule priority, isActive (rule enabled or not). For a more extensive description, please refer to [15, 16].

The following sections describe the event specifications and rule syntax of REFLEX knowledge.

Event Specification

The *ON* or *event* specification clause of the rule allows both *primitive* (simple) or *complex* (compound) events to be specified. The event clause is expressed using an event algebra. The event algebra expresses the temporal relationship between component events. Event chronologies or histories need to be maintained in order to satisfy the event clause. This is the primary purpose of the *temporal log* [15]. The time model employed is based on *interval logic*, with all events having a before/after granularity. All internal events (described later) are preceded by either a *before* or *after* statement. If no mention is made, then *after* is assumed.

A list of the different types of events follow: (i) Internal: *Object events*: before/after create, get, put, delete. *Transaction events*: before/after start, commit, abort (ii) Temporal events: at (specific-time), periodic (repeat-after-period), after (duration), sequential (iii) External events: These events are application defined and hence cannot be listed.

The algebra contains several logical and temporal keywords. The logical keywords are **AND, OR, XOR** and **NOT**. The temporal keywords are **BEFORE, AFTER, PRECEDES, SUCCEEDS, AT, BETWEEN, ON, WITHIN HOUR/MIN/SEC, EVERY HOUR/MIN/SEC, MIN, MAX, DATE, TIME**. Parenthesis are used to override operator precedence.

Events are augmented with the concept of *validity*. An event is only a valid event if it occurred within a certain period. The temporality of events can be quite diverse. Examples of the *user-friendly natural english* event clause syntax are:

Event$_1$	simple single event of any valid type
Event$_1$ **AND** Event$_2$ **WITHIN MIN** 30	both Event$_1$ and Event$_2$ must occur within thirty minutes of each other
Event$_1$ **PRECEDES** Event$_2$ **WITHIN HOUR** 24	as above, but in sequence
AT TIME 17:00	

Events are totally *user-definable*. Internal events are provided by REFLEX, as are clock-based events. External (application based) and any other types of event, are defined by the user or designer of the active application. All events are detected by the Event Manager/Detector. On detection, the events are first logged in the temporal log, and then the Knowledge Management Kernel is informed of their occurance.

Rule Syntax

Being an adaptive and portable data model, REFLEX allows its condition or action clauses to be expressed in either generic Object SQL or in the proprietary language of the host DBMS. For purposes of illustration, the syntax for REFLEX rules is as follows:

ON	event-algebra			
IF	either	i) no condition	ii) Object SQL query	iii) proprietary lang. query
THEN	either	i) *call* user module	ii) Object SQL query	iii) proprietary lang. query

Each clause is further exemplified:

i) *event algebra clause*. As described earlier in the section on Event Specification, the clause may be complex or primitive. The clause is subject to our event algebra.

e.g.

update(table/class/object name) on a update or modification of a table, class, or instance object

update(table/class/object name) AND TIME 5.00pm

The event algebra expression provides for a powerful mechanism for testing that certain events have taken place, or are likely to take place, without sacrificing effiency by testing the rules condition clause.

ii) *condition clause*. The clause may be set to TRUE (i.e. no condition), or expressed in either Object SQL or the host DBMS"s proprietary language.

e.g.

TRUE i.e. no condition, just an Event-Action pair.

or

class_name.attribute (=, <, <=, >, >=, !=) *expression*

```
IF      SELECT    a.Name(), a.Salary(), b.Name(), b.Salary()
        FROM      employee a, employee b
        WHERE     a.Name() = OBJECT1
            AND   a.salary > b.manager.salary
```

The above encodes the familiar constraint that an employee cannot earn more than his/her manager. A generic Object-SQL dialect has been employed, for the IF or condition clause, to allow for portability between platforms.

iii) *action clause*. The clause may *call* an object module or as, with the condition clause, a more complete query may be expressed in either Object SQL or the host DBMS"s proprietary language.

e.g.

call program module <argument list>

or

delete object

Some examples rules follow:

```
ON      update account
IF      select    c.Name()
        from      account a, customer c
```

```
ON      update stock
IF      select    s.itemNo()
        from      stock s
```

```
where    a.customer.name = OBJECT1
    and a.customer.name = "Fred Bloggs"
    and account.credit_limit = 0
THEN call Alert_No_Credit
```

```
where    s.itemName() = OBJECT1
    and s.itemQty < = s.reorderQty
THEN insert(onorder) itemNo, reorder_qty
```

Adding Rules to the System

The user or developer of an active application, using the REFLEX active extension, is presented with a fully object-oriented graphical user interface (GUI), the **REFLEX Visual Supervisor** (VIS). At present rules and events are added and declared to the REFLEX system by means of the VIS. The user interface and access methods of REFLEX are the subject of another paper, currently under preparation. A brief introduction to the prototype is presented.

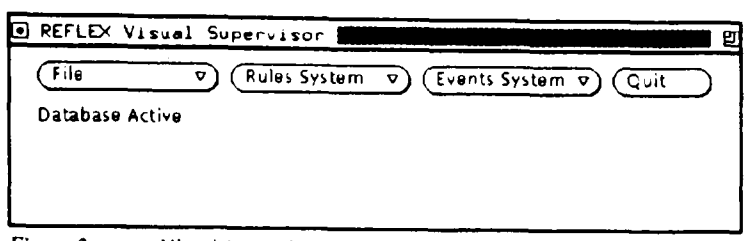


Figure 2 Visual Supervisor Menu

The user is presented with the simple *REFLEX Visual Supervisor* menu. If the user wishes to perform system administration functions i.e. add new rules or events, or to interrogate the system, they are all completed using this module.

When a user wishes to add a new rule, the *New Rule* option is selected from the Rule System menu. The user is presented with a *New Rule Dialogue Window*. The name and description of the new rule are entered. Following this, the user is presented with a *Class List Window*. This window shows all the classes or tables that are available to the user"s application. From this list the user drags the class or classes from the window, onto the Rule Dialogue Window, that the rule will affect. A further window is presented, with the instances of all the classes selected. The user is then able to select particular instances as targets for the rule"s action or the user is able to highlight any *exemptions* from the rule"s action.

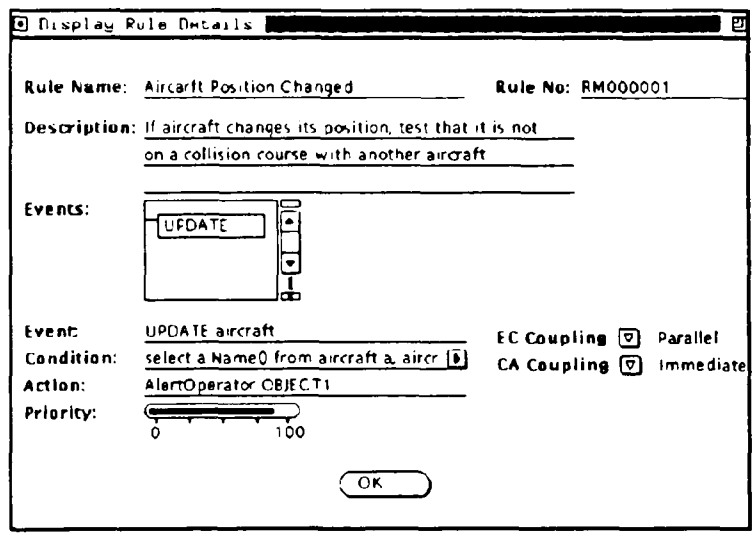


Figure 3 VIS Display Rule Details Window

After the classes have been selected, the event algebra clause must be specified. This is accomplished in much the same fashion as the class selection, except that when the required events have been selected from the *Event List Window*, the user is able to select the logical and temporal operators, to complete the event algebra.

The condition clause is completed semi-automatically, as the user is presented with windows of all classes, with sub-windows showing the attributes and/or methods of the classes (if any). The user must (with our current prototype), enter the remainder of the query. The system checks for syntactical and existence errors at this stage.

The action clause is completed in the same way as the condition clause, but using windows, which list the available programs and objects which may be selected for the program calls. More extended query language statements are available for the action statement than are available for the condition clause. This is because the action clause may include query operations such as insert or delete.

Both the Condition and Action clauses can access the object that raised the event by using the keyword *OBJECT* followed by its occurance number in the event algebra expression. In the figure above, the action clause calls a user defined operator window called *AlertOperator* and passes the *OBJECT1* as an argument, which in this case will be the actual aircraft object that has been updated.

The same type of approach is used for event management, i.e. decalration of new events etc. The VIS system also provides a rule browser.

2.2. Distributed Systems

The design of REFLEX using object-oriented technology makes it possible to adopt a *distributed* and *parallel* implementation model. This is possible as the modules within REFLEX are modelled as objects. The objects are *autonomous* and communicate with each other using *messages*. Thus the objects can execute on separate processors independently of other processes, on the same multi-processor machine, or as processes on a client-server distributed system. Thus *objects are natural models of concurrency* and exhibit *client-server* communication.

The REFLEX architecture is implemented as a parallel and distributed system i.e. all the modules are executing concurrently. There are areas of difficulty which may be subject to parallelisation. An example of such a tentative concurrent execution could be as follows. If an event has been raised, both the Knowledge Selection Module (KSM) and Condition Evaluation Module (CEM) execute simultaneously trying to satisfy their event algebra and condition clauses respectively for the same rule, see figure 4. As a result, the condition clause is evaluated (by the CEM) and possibly satisfied by the time the event clause has been evaluated (by the KSM). Normally, the condition clause is not evaluated until the rule has a state of *event-clause-satisfied* i.e. until the event clause has been satisfied.

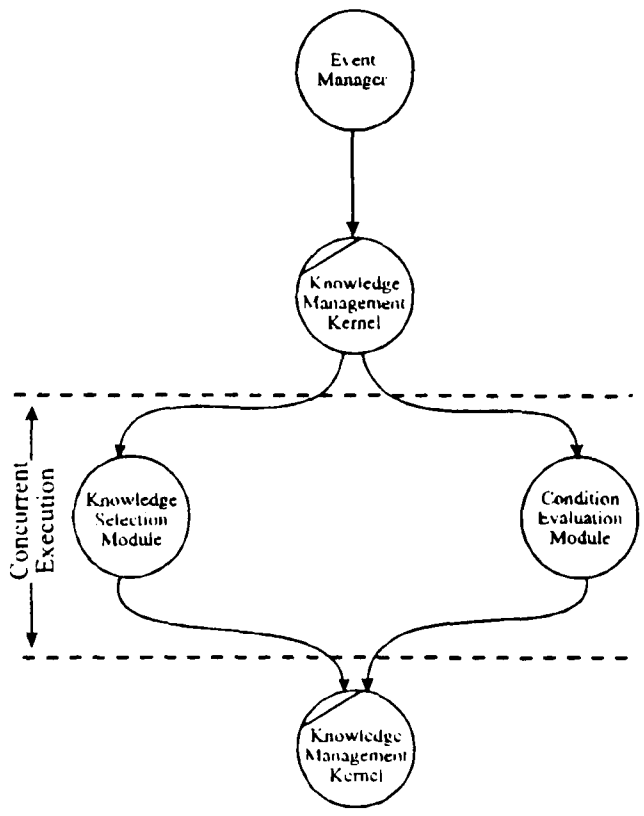


Figure 4 Concurrent Execution KSM & CEM

This feature is desirable in *critical real-time* situations. It may only take place for high priority *trap* rules, at the application designers discretion. However this feature may not always be desirable as efficiency of the database may be adversely affected. But by using the parallelism only for critical, high-priority periods, it could improve the response time of the overall system without overloading the system performance at normal periods by causing unnecessary condition clause evaluation. However, this, as

indicated above, is an efficiency decision made by the designer.

Thus it enables the construction and implementation of high performance systems, capable of modelling real-time critical applications.

2.3. Employing Activity

An active database provides a very fast reaction to any changes within the database"s state or the applications environment i.e. *imparting* active capability into the application domain. REFLEX, unlike any other active database research prototypes as far as we are aware, *employs* the active capability itself i.e. it is *self-active*. The knowledge base (KB) as well as an application database are stored within the REFLEX system. Thus the maintenance of the KB can also be subject to the notion of activity. As an example, the rule"s state is monitored *actively* by the REFLEX system. Rules have three components: events, conditions and actions. The clauses for each of these components are compiled, translated or recompiled at the point of rule creation or on rule modification. The re-compilation process being automatically triggered on a rule change.

2.4. Non-Destructive Knowledge

REFLEX introduces the concept of *Non-Destructive Knowledge*. By this we mean that if a rule has been declared, and it has not been used, it may be subject to change or amendment. But if the rule has been fired, or linked, it may no-longer be subject to change. It is in effect, locked. This concept allows us to audit our knowledgebase and evaluate why certain events occured. It also allows the provision of *knowledge versioning*. If a change in the rule"s definition is required, a new rule must be declared, which the old rule references. The rules even if deactivated, still maintain references to objects that they refered to, thus providing a browsing system of the previous database knowledge state.

2.5. Rule Contention

A conflict resolution mechanism has been implemented as a task of the KMK as part of its scheduling process. The operation of rule conflict resolution depends on the assignment at knowledge capture time of priorities to the object rules. By default, a rule"s priority is zero unless it is given a different value by the designer or user. The priority for rules may take any values the designer wishes, but a default is between 0 and 100. If two rules have the same priority, the first to be detected and hence obtain a state of "*in context*", is actioned first. If the rules have a *trap* priority, then they are executed in parallel.

As explained earlier and as a further example of the "*self-active*" notion mentioned above, when the rule"s priority changes, the system will automatically recompile the rules actively. This is a useful and desirable feature since it makes for a flexible system that can easily respond to change of knowledge or rule by the end-user without a need for a developer"s intervention.

3. Related Research

Research into active database systems is intense. There are many research groups building research prototypes around the globe. The major research in active databases includes the following prototypes: POSTGRES [20, 21], STARBURST [10], HiPAC [1, D1, 3], ADAM [18, 4] and ODE [6]. REFLEX differs from these research prototypes by its *loose-coupling* to the underlying model in contrast to the high degree of coupling the other models employ. A more in-depth survey can be found in [13]. Another major area that is novel, is that the REFLEX model is designed to function concurrently with the host DBMS. Its design encompasses a multi-processor distributed architecture.

4. Conclusions

REFLEX has proven to be a very effective research prototype. We have implemented the REFLEX extension onto ONTOS [17], on Sun Solaris[23], using AT&T"s C++ v2.1[22]. The REFLEX system *has been demonstrated* at various venues, using a graphical simulation of an Air Traffic Control System (see appendix). We are now using REFLEX to generate data on how a *real active applications behave*. We are currently implementing REFLEX onto POET [9], under Microsoft (MS) Windows v3.1 using MS C++ v7.0[12]. This will demonstrate the adaptability and portability features of the model. As REFLEX has been implemented to assertain its viability, performance whilst a major issue, was not a priority for the first prototype. Performance enhancements have been designed in the model, but have not been realised at this early stage. REFLEX has introduced a number of novel features such as its non-destructive knowledge model, its critical concurrency appoach and its self-activity to name a few.

The use of object technology has allowed the construction of a model that closely resembles the real world scenario. Major benefits of object technology have been utilised in terms of distributing the executing processes of REFLEX to both multi-processor and client-server architectures. The fact that the objects can very easily be mapped to independent processors as they exhibit autonomous behaviour via encapsulation, have well defined interfaces, and communicate with each other via messages; they have proved themselves as *natural models for concurrency*.

The knowledge representation schemes of both production-rules and frames together allow REFLEX to support cause/effect and deep knowledge reasoning. The productions providing the cause/effect knowledge and the objects the deeper knowledge about the application domain.

REFLEX is designed as a general purpose active database extension, but its operating tolerances cover the spectrum from critical real-time systems where immediacy is a major concern, and other systems such as stock control, where immediacy of response is important but not critical. For systems where safety, is a major concern, the system must have been carefully validated and verified so as to guarantee a high degree of safety. We are presently working on validation and verification of the REFLEX model and architecture, using petri-nets. We are developing a case tool PETENG which will allow the automatic verification of any petri-net structure, against a number of dimensions. Currently REFLEX utilises the rule-

set concept to minimally validate and control its knowledge content. We are currently working to provide a much safer method of rule-set construction, using both static and some novel dynamic analysis techniques.

We intend to make REFLEX available on the public-domain, via ftp. It will initially be released for the ONTOS DBMS system. Please contact the authors for further details.

5. References

[1] Chakravarthy S., Blaustein B., et al, "HiPAC: A Research Project in Active, Time-Constrained Database Management", Final Technical Report, Xerox Advanced Information Technology Division, July 1989

[2] Dayal U., Blaustein B., et al, "The HiPAC Project: Combining Active Databases and Timing Constraints", ACM Sigmod Record, Vol. 17, No. 1, March 1988

[3] Dayal U., "Active Database Management Systems", Sigmod Record, Vol. 18, No. 3, 1989

[4] Diaz O. and Paton N. W., "Sharing behaviour in an object-oriented database using a rule-based mechanism", Proc. 9th British National Conference On Databases, 1991

[5] Dittrich K. and Dayal U., "Active Database Systems", Tutorial Notes, VLDB 91, Barcelona, Spain, September 1991

[6] Gehani N.H. and Jagadish H.V., "Ode an as Active Database: Constraints and Triggers", Proc. 17th Int. Conf. Very Large Data Bases, Barcelona, September 91

[7] Gehani N.H., Jagadish H.V. and Shmueli O., "Event Specification in an Active Object-Oriented Database", Proc. 1992 ACM SIGMOD Intl. Conf. on Management of Data

[8] Gehani N.H., Jagadish H.V. and Shmueli O., "Composite Event Specification in Active Databases: Model & Implementation", Proceedings of the 18th Int. Conf. on Very Large Data Bases, Vancouver, Canada, 1992

[9] Gwb, "P.O.E.T. Reference Manual v.1"

[10] Lohman G. M., Lindsay B., Pirahesh H. and Schiefer K. B., "Extensions To STARBURST: Objects, Types, Functions, and Rules", CACM October 1991, Vol 34, No 10

[11] Minsky M., "A Framework for Representing Knowledge", The Psycology of Computer Vision, McGraw-Hill, 1975, pp. 211-277

[12] Microsoft, "MS C/C++ v7.0 Programmers Manual", MicroSoft, 1992

[13] Naqvi W. and Ibrahim M.T., "The REFLEX Active Database System", Database Systems Research Laboratory, Technical Report TR-CIT-DB0692, University of Greenwich, 1992

[14] Naqvi W. and Ibrahim M.T., "REFLEX: An Active Database Extension", Poster at the 11th British National Conference on Databases, July, 1993

[15] Naqvi W. and Ibrahim M.T., "REFLEX Active Database Model: Application of Petri-Nets", Proc. of the 4th Int. Conf. on Database and Expert Systems Applications, Prague, September 1993

[16] Naqvi W. and Ibrahim M.T., "REFLEX: An Active Object-Oriented Database Model", sumitted

for publication, April 1993

[17] "ONTOS Reference Manual", ONTOS Inc, 1991

[18] Paton N.W., "ADAM: An Object-Oriented Database System Implemented In Prolog", Proc. 7th British National Conference On Databases, 1989

[19] Ringland G., "Structured Object Representation - Schemata and Frames", Approaches to Knowledge Representation, Ed. Ringland and Duce, 1987, pp 81-99

[20] Stonebraker M., Hearst M. and Potamianos S., "A Commentary on the POSTGRES Rules System", Sigmod Record, Vol. 18, No. 3, September 1989

[21] Stonebraker M. and Kemnitz G., "The POSTGRES Next-Generation Database Management System", CACM October 1991, Vol 34, No 10

[22] Stroustrup B., "The C++ Programming Language", Addison Wesley, 1986

[23] Sun Systems, "Solaris 1.1. User Manaual", 1992

[24] Widom J., Cochrane R. J. and Lindsay B. G., "Implementing Set-Oriented Production Rules as an Extension to Starburst", Proc. of the 17th Int. Conf. on Very Large Data Bases, Barcelona, Spain 1991

Applied Active Databases for Evolving Image Processing Algorithms

W. Naqvi and S. Panyiotou

School of Computing and Mathematical Sciences
University of Greenwich, London, SE18 6PF, U.K.
w.naqvi@greenwich.ac.uk
http://www.gre.ac.uk/~nw01/reflex

Abstract

To develop an algorithm for any application takes thought and a lot of trial and error. The algorithm must be coded, compiled, tested for compliance with the specification. If it does not perform to target, the code must be amended, recompiled and tested again. The process is cyclic and time consuming. In this paper a novel method is introduced which allows the building or tuning of algorithms or programs at run-time by using an active database. The paper uses the domain of robotic vision as a case study to introduce the concept, particularly the first stage of the object recognition process known as ***segmentation*** *i.e. extracting the primitive characteristics of the objects of interest. The system has been implemented upon the REFLEX active database system.*

Keywords: active database, mutating algorithms, active algorithms, segmentation algorithms, image processing, knowledgebase systems, active application, REFLEX

1. Introduction

Many industrial applications require some sort of computer interpreted vision systems e.g. the automatic transport trolleys in car manufacturing plants or robotic quality control inspectors. In the case of robotic quality control inspectors, the robots would look out onto a conveyor belt of finished products and would reject any products that do not match the quality control requirements. However, this sort of application has some intrinsic problems; the robot would require human-like stereo vision so that a determination of depth could be perceived, very important in quality control. The computer would be required to process and identify the objects within the scene and determine whether they match the quality control

requirements, all in real-time.

A very important stage of the object recognition process is to reduce the object to its primary components. At this point the basic characteristics of the object are determined such as number of vertices, number of lines, closed regions etc. This process is called ***segmentation***. The original image is split into ***regions***, which we hope represent surfaces in the real world from which the image came. The purpose of segmentation is to pass onto subsequent algorithms a symbolic representation of the scene. As the objects are segmented, they are matched against a database of known objects.

This paper introduces a new method for dynamic algorithm creation and tuning by providing a novel use for active databases, namely that of providing *active algorithms*. This work is reported in the context of robotic vision systems as this is the first application area. Active database technology is used to automate the segmentation process.

Active databases have two main tenets i) they encapsulate an application's domain knowledge within the system and ii) they use the event-driven paradigm. The philosophy of the REFLEX [10, 11], an active database system, architecture and design encompasses the provision of a flexible, adaptive and *active* capability to an organisation's existing database. Most active prototype systems use the notions of event-condition-action (ECA) knowledge model as described by McCarthy and Dayal [9]. REFLEX has an enhanced knowledge model, the EECA [12] which promotes new ideas such as redundant action and fail-action clauses, user-definable coupling mode dependency, short-circuit evaluations to name but a few.

Traditionally the selection of segmentation algorithms has been domain specific and appropriate algorithms may not have been utilised, resulting in slow and inefficient segmentation. More programming effort is needed at later stages to rectify the inefficiencies.

This paper reports on an automatic segmentation design system known as the **Cortextual Parser** (CP), that is not scenario dependent. The central component of the CP is the REFLEX active database system. REFLEX allows for the maintenance, mutation and optimisation of segmentation algorithms, dynamically at run time. In contrast to traditional programming systems which are inherently static and inflexible.

The paper is organised as follows, section 2 summarizes the REFLEX active database model. Section 3 gives an overview of image processing and segmentation techniques. Following on section 4 describes the data model employed for the CP. The architecture for the CP is overviewed in section 5. Finally section 6 concludes with future directions.

2. Reflex Summary

Much research into active databases is being conducted such as HiPAC [3], StarBurst [21], ADAM [5] and REFLEX [10]. REFLEX is unlike any of the other mentioned research prototypes since it is *loosely-coupled* to its underlying database. It has been designed to be portable and adaptive to any new commercial host DBMS and has been successfully implemented upon both ONTOS [15] and POET [16]. The related research prototypes have not been afforded this flexibility as the active features have been built into the databases either from the outset or later directly into the source code. REFLEX has been designed as a fast reacting real-time database system [10, 13]. It has proven itself ideal for the application described in this paper, as it can respond *reflexively* to any changes within the environment. REFLEX also allows the ***active*** notion to play a dual role in the prototype. The first role, which is common to other prototypes, is *imparting* active capability into the application domain. The second role is *employing* active capability itself, e.g. if a new rule is added, the knowledge representation is restructured to reflect the change i.e. the system is *self-active*.

REFLEX is a portable active database extension and as such provides a powerful event specification language called the English ESL, a query facility based on a high level object SQL and parameter passing from the ESL to the OSQL query using the OBJECT keyword as a parameter referencer. These features of the REFLEX active extension, its architecture and others, have been reported on before, the interested reader may refer to [11, 12].

The following section introduces the application domain of image processing.

3. Image Processing

Image processing is a set of techniques that enhance a source image. These include noise reduction, edge detection and light equalisation. Image processing is a primary step before an image can be *segmented*. The techniques involve convoluting an image with some sort of filter dependent on the result required. Image processing does not extract any object information from the scene [22]. The main problem with existing segmentation algorithms are their poor performance on a set of images different from the ones that were used in their initial development. Changes in the image can be defined in terms of content and quality. This limitation has created a major bottleneck in most vision systems. We thus need new algorithms where the parameters of the segmentation algorithms are a set, dependent on the global image, target characteristics, and contextual scene information. However, experiments have shown that the performance of the segmentation algorithms cannot be improved beyond a limited domain by only adjusting their parameters. The reason for this is that many of the basic assumptions made in design of these algorithms are violated when different scenarios are encountered.

An attempt at improving segmentation has been the knowledge-based segmentation approach, such as that proposed by Sadjadi and Nasr [17]. The techniques attempt to select the appropriate segmentation algorithms from a library of two or three existing algorithms. However, they also suffer from the same problem as the first techniques. This is due to the fact that the algorithms work in a narrow domain, and even in their own domain, their performance is unstable. Most of the existing available algorithms have a large area of overlap, hence are inefficient. Again, all these algorithms fail dramatically in many instances because they are based on fixed assumptions which in many cases are not valid. Consequently selecting among a set of given algorithms cannot lead to a meaningful improvement in performance.

If the computer vision problem were merely to recognise or classify a scene from one of several candidates, then we could employ special purpose hardware such as the WISARD system [2]. This system can be trained in literally a few seconds to discriminate between a small set of different scenes. Its discrimination abilities far outweigh conventional vision systems in terms of speed. The problem with the special hardware approach occurs if discrimination between larger sets of scenes is required. When there are several objects, each in any orientation and at any relative position, combinatorial explosion is rapidly encountered.

The subject of segmentation algorithm selection and design needs to be addressed, whereby the algorithms design is a planned sequence of different image processing and segmentation primitives. Information is needed on the image for this process to be efficient. These include the following:

a) desired goal (find all resistor < 5mm long)

b) the input images and target matrices (contrast, signal to noise ratio etc)

c) contextual knowledge (targets are closed regions and lines)

REFLEX has been employed to allow an adaptive and evolving mechanism for the automatic design of special purpose active, high performance segmentation algorithms. Different scenarios will lead to the generation of different plans, based on the knowledge available from the scene domain within REFLEX. Figure 1 shows the basic concepts described above.

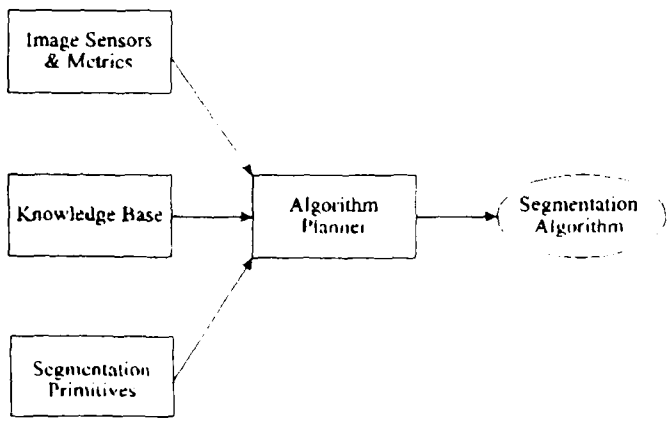


Figure 1 Concept diagram for automatic segmentation design

Work in the general area of image processing and automatic design has been very limited. De Hass [4] presents a system that can generate programs for machine vision systems that measure several parameters of an industrial object in a video scene. Iwase [7] describes an expert system for image processing. Their technique is an interactive process whereby the user provides a set of answers

about the problem such as the goals, image characteristics, etc. In response to a fixed set of questions. The problem with this is that the user interprets the scene, which may be incorrect. The system does not attempt to analyze the scene in any way whatsoever. The rules are static and thus not applicable to many slightly differing domains.

4. Data Modelling

When an image requires segmenting, the developer selects an algorithm from a handful of algorithms that are available in the current literature. These segmentation algorithms are static and once coded, do not change. Even though the selected algorithm works well in the scenario for which it was first created, it may not be as efficient for the current scenario.

The **Cortextual Parsing** system described in this paper, automatically evaluates and primarily parses the new domain image. From the results of the parsing stage, an initial segmentation plan is generated which comprises of an algorithm or set of algorithms required to segment the image. As the system executes the plan, it monitors the execution, against any critical temporal constraints, and also any conditions dependent on the domain. The result of the monitoring, may require that no action is taken, or if the system decides that the current algorithm is inefficient, it either *mutates* the algorithm or replaces it with a more probable algorithm and the monitoring process, starts again.

Even if the processing of the algorithm performs within the bounds of a time period, the algorithm is continually monitored, and is subject to mutation if the systems intelligence unit considers that it can be made even more efficient for the current domain i.e. it is tuned further. As the number of mutations increase over time the efficiency of the algorithm increases and peaks until further mutations cause little or no change.

How can evolving and mutating algorithms be modelled? The approach adopted in this research is a coupling of an active database to a planner, described later in section 5 on the architecture. The active database maintains a store of image data, image processing and segmentation primitives, segmentation algorithms as well as the knowledge required to successfully plan and produce efficient segmentation algorithms.

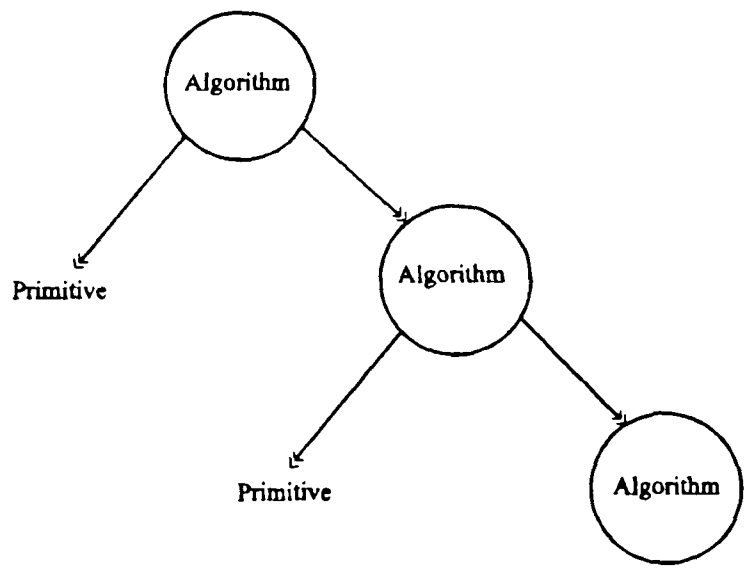


Figure 2 Nested Algorithm Objects

The data model employed is object-oriented where the basic unit is that of an algorithm object. This is a complex object, as it contains aggregates of the segmentation primitives (that may work for specified small regions and other higher-order algorithms), so it is essentially recursive, figure 2. This approach allows REFLEX to execute an algorithm and monitor its progress. If the monitoring system discovers that the algorithm is under-performing it can swap it out, be it a sub-algorithm, primitive or parameter and replace it with another item of the same or differing type i.e. the system tunes itself to use the most efficient algorithm given an approximately known scenario. It is thus an extremely adaptive system and not domain specific.

During the segmentation phase, the results of each of the algorithms and its composite primitives are stored. The active database maintains the algorithms and modifies the algorithm's structure to hold required primitives and adjusts any threshold parameters. The system can be explained by means of an example application, such as a robotic quality control for printed circuit board (PCB) fabrication. A PCB production unit fabricates many PCBs. There are many levels of quality control. A robot is assigned the task of inspecting a board, whilst on the conveyor belt, quickly and repetitively. This inspection first involves evaluating the overall size and shape of the PCB. An example active rule could be:

```
ON   EVENT SCAN pcb
IF   SELECT  s.Name()
     FROM    shape s, shape p
     WHERE   s.Name() = OBJECT1
          // OBJECT1 is the event parameter
     AND     s.Name() = p.Name()
     AND     (s.SHAPE < > p.SHAPE  OR s.TIME > p.CRITICAL.TIME)
THEN   CALL rejectScan(s)
```

This rule is invoked when the event SCAN is raised against a pcb object. The object *s* is set to the scanned object by use of the OBJECT1 identifier, which is REFLEX's mechanism to reference the object that raises an event (the number after the keyword OBJECT refers to the order of the event). The object *s*, has member function SHAPE, which returns the overall shape of the visual object. The object *p* is the reference object held in the database. The database is pre-loaded with segmentation algorithms, that apply to different goals or queries. As the SHAPE function attempts to segment the image and return its outline shape, it is actively being monitored (against the reference), so that it uses the best available segmentation algorithm to return the shape. If the algorithm is not efficient enough, it can be replaced by another algorithm from the database (of the same type), or it can be *mutated* to generate a new segmentation algorithm.

```
ON      EVENT rejectScan
IF      NULL
THEN    call mutateAlgorithm(OBJECT1)
```

0

For the above high level rule, the system must provide a segmentation algorithm that returns the shape of the object, in this case a PCB, but within a specified time-frame. Mutation rules could be:

```
ON   EVENT SCAN pcb
IF   SELECT s.Name()
     FROM    shape s, shape p
     WHERE   s.Name() = OBJECT1
     AND     (s.TIME > p.CRITICAL.TIME
     ANDs.TIME < = p.CRITICAL.TIME+5%)
THEN    s.SCAN.ALGORITHM.
        ADJUST.PARAMETER(1%)
        AND  COMPILE, SCAN


ON   EVENT SCAN pcb
IF   SELECT s.Name()
     FROM    shape s, shape p
     WHERE   s.Name() = OBJECT1
     AND     ( s.TIME > p.CRITICAL.TIME
     AND s.TIME > p.CRITICAL.TIME+5%)
THEN    s.SCAN.ALGORITHM.
           SWAP.PRIMITIVE()
           AND  COMPILE, SCAN
```

The above two rules would either adjust the parameter to a primitive of the scanning segmentation algorithm, if the scan time just exceeded the critical time by upto 5% or if the difference was greater than 5%, the primitive would be swapped out and replaced.

After the correct shape and size have been determined, further high-level goals could be tried, such as, are the components in the correct positions, i.e. are the links correct. In order to evaluate such goals, e.g. how many resistors are on the PCB?, the system interrogates the matrices database to determine what properties a resistor possess, e.g. what size should it be, what shape it has, what the coloured rings mean. Armed with this information, the system is able to satisfy queries such as: return all resistors < 200 ohms. An example high-level rule:

```
ON   EVENT SCAN pcb
IF   SELECT s.Name()
     FROM    shape s, shape p
     WHERE   s.Name() = OBJECT1
     AND     ( s.R1.POSSTART < >  p.R1.POSSTART
     OR  s.R1.POSEND < >  p.R1.POSEND)
THEN    CALL REJECT (S.R1, POS)
```

For the above rule, on the scan being performed, the system will check that the resistor being inspected is in fact inserted in the correct position on the PCB.

5. CP Architecture

Using REFLEX the system can process multiple images concurrently since the system has been designed as a fast, real-time database. A block diagram of the design system is shown in figure 3. The input to this

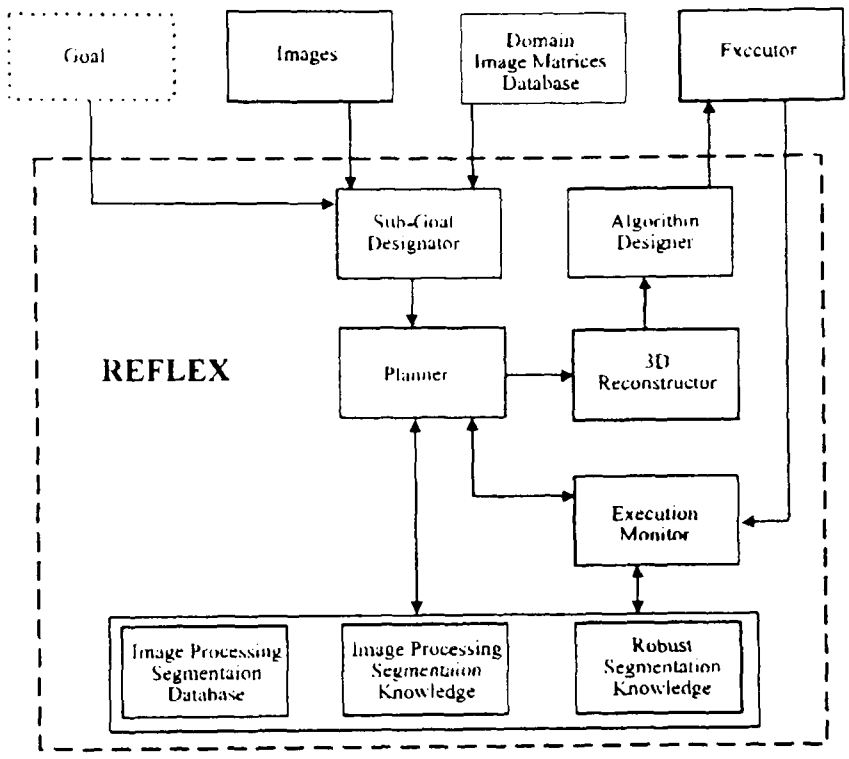


Figure 3 Cortextual Parser Architecture

system would be the following.

a) a set of slightly different images from a number of sensing devices, stereopsis. This allows 3D information to be extracted from the segmented objects

b) a set of target metrics. This takes the form of a basic description of the object, in terms of lines, circular regions etc. This information allows the user to develop algorithms that are application specific. The information may be stored in a database (as matrices) for retrieval by the system when a goal is selected

c) image metrics are also calculated this includes things like frequency of edges, signal-to-noise ratio, light intensities in the image, etc. Special purpose hardware can be deployed to aid in this process.

d) finally an explicit segmentation goal is entered (Find all resistors < 10 mm AND > 200 ohms).

The basic output from the system is a segmentation algorithm specification that is a collection of sequential

image processing steps. The designed algorithms are tailored to a particular scenario. For each different scenario a different plan is created.

In the planning, we start with a set of initial states and desirable states (goal). We then attempt to devise a plan that can achieve the goal [20, 6]. The initial state in this case is the set of raw images. The goal is a desired segmentation result that can be expressed at different levels, such as: find resistor, find all resistors < 200 ohms, etc. The sub-goals are then extracted by REFLEX, which has the specific domain knowledge for the particular application. The knowledge-base contains information about which operations to perform given certain images. The knowledge-base is represented as a rule-base structure, within REFLEX.

Traditionally, planning in AI has not been done using explicit knowledge-base (like an expert system), but rather using very primitive knowledge (expressed in first order logic) and theorem proving techniques. The knowledge-base contains two types of information: i) detailed knowledge about the image processing primitives, when they can be applied, their limitations, and how a collection of them can be applied to a given scenario (plan of execution) and ii) knowledge about how the parameters of a specific image processing algorithm can be tuned for a given application.

The robust segmentation knowledge base holds algorithms that will stop the image processing and segmentation routines from falling over if the image is not in a perfect condition. This knowledge is held as production rules. The parameters of the rules are modified by REFLEX depending on the image characteristics. For example if the image appears to have strong features then a threshold of three pixels may drop to one or two, alternatively it may rise to five or six pixels if the image has weak characteristics. The characteristics are determined by the CP on the first scan and are placed in the image matrices and subsequently referenced at a later stage.

The idea behind the sub-goal designator in REFLEX, is to create a set of sub-goals for the system dependent on what the ultimate goal is. Thus a different set of sub-goals are generated dependent on the main goal. Each sub-goal is achieved through a set of image processing primitives. These matrices are extracted depending on the desired goal. The matrices include vision information such as lighting conditions, signal-to-noise ratio, object entropy and target range. These matrices are modeled as complex objects which contain aggregates of further lower-level matrices and of simple primitives such as lines, colours, size etc. This information, stored within REFLEX, maybe on the change of conditions or objects.

6. Conclusions

The work reported in this paper, describes a new and advanced way of using active databases to provide

dynamic programmability. The programs and algorithms can be changed during run-time, even though the host programming environment is essentially static. The system is highly adaptive and modifies itself to produce an optimum solution. This approach allows active databases to enter the areas or preserves traditionally held by genetic algorithms. The REFLEX active database system has proven a robust vehicle for testing the active algorithm concept described within this paper.

The **Cortextual Parsing** system described provides automatic and tuned design of segmentation algorithms, which would simply be inserted to the visual target system, whether it be quality control or a full-blown image understanding system. For larger systems, where the domain is extensive and largely unknown, then the whole of the Cortextual Parser could be included to provide, at run-time, segmentation algorithms for unknown domains. The system is extensible, as new segmentation and image processing primitives are developed, they may be easily added to the database along with the knowledge of when to apply such primitives. The Cortextual Parsing system has currently been implemented for an industrial PCB Fabrication Quality Control system, on an earlier prototype of the REFLEX ONTOS platform, on Sun Workstations and XWindows. We envisage that further domains may be prototyped in the future, by simply changing the image matrices to reflect the new domains. These domains will be subject of further active algorithm prototypes using REFLEX's EECA knowledge model [12].

We believe that this research will establish the feasibility of using active databases to provide *mutating* **active algorithms** (in this case segmentation algorithms) over time, for a variety of different domains.

7. References

[1] Agrawal R. and Gehani N.H., "Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++", 2nd Int. Workshop on Database Programming Languages, Portland, OR, June 1989

[2] Aleksander I., Thomas W.V. and Bowden P.A., "WISARD - A Radical Step Forward in Image Recognition", Sensor Review. July 1984

[3] Chakravarthy S., Blaustein B., Buchmann A., et al., "HiPAC: A Research Project in Active, Time-Constrained Database Management", Final Technical Report, Xerox Advanced Information Technology Division, July 1989

[4] De Haas L.J., "Automatic Programming of Machine Vision Systems", Proceedings of the International Joint Conference on Artificial Intelligence, 1987

[5] Diaz O., Paton N.W. and Gray P., "Rule Management in Object-Oriented Databases: A Uniform Approach", Proc. of the 17th Int. Conf. on Very Large data Bases, Barcelona, Spain, 1991

[6] Genesereth M.R. and Nilsson N.J., "Logical Foundation of Artificial Intelligence", Los Altos, California: Morgan Kaufmann, 1987

[7] Iwase H., Toriu T. and Gotoh T., "An Expert System for image processing", Proceedings of the Fourth

Conference on Artificial Intelligence Applications", San Diego, March 1988

[8] Marr D., "Vision", Freeman, San Francisco, 1982

[9] McCarthy D.R. and Dayal U.,"The Architecture of an Active Data Base Management System", Proc. ACM SIGMOD Intl. Conf. on Management of Data, Portland, June 1989

[10] Naqvi W. and Ibrahim M.T., "REFLEX Active Database Model: Application of Petri-Nets", Proc. of the 4th Int. Conf. on Database and Expert Systems Applications, Prague, September 1993

[11] Naqvi W. and Ibrahim M.T., "Rule and Knowledge Management in an Active Database System", Proc. of 1st Int. Workshop. on Rules in Database Systems, Edinburgh, September 1993

[12] Naqvi W. and Ibrahim M.T., "EECA: An Active Knowledge Model", Proc. of the 5th Int. Conf. on Database and Expert Systems Applications, Athens, September 1994

[13] Naqvi W. and Ibrahim M.T., "Active Distribution by Stealth", Proc. of the 6th Int. Conf. on Database and Expert Systems Applications (workshop), London, September, 1995

[14] Naqvi W., Panayiotou S., Soper A. and Ibrahim M.T, "Cortextual Parsing: The use of an active database to provide semi-evolving segmentation algorithms", Tech. Report CIT-DSRL069301, University of Greenwich, June, 1993

[15] "ONTOS Reference Manual", ONTOS Inc, 1991

[16] "POET 2.1 Programmer's & Reference Guide", POET Software Corporation, 1994

[17] Sadjadi F. and Nasr H., "A technique for automatic design of image segmentation algorithms", Proceedings of the SPIE - The Int. Society for Optical Engineering, Vol: 1098 p.177-81, 1989

[18] Stonebraker M. and Kemnitz G., "The POSTGRES Next-Generation Database Management System", CACM October 1991, Vol 34, No 10

[19] Subbarao M., "Interpretation of Visual Motion: A Computational Study", Morgan Kaufmann Publishers, 1988

[20] Wilensky, "Planning and Understanding", Reading, Addison Wesley, 1983

[21] Lohman G. M., Lindsay B., Pirahesh H. and Schiefer K. B., "Extensions To STARBURST: Objects, Types, Functions, and Rules", CACM October 1991, Vol 34, No 10

APPENDIX B

Example Application Runs

This appendix provides runs from example applications from the two considered domains of (i.) Air Traffic Control Systems, and (ii.) Student Records systems.

1. Air Traffic Control System

```
>>nw01@splinter 101 % make
cd .. ; make
echo returned
returned
main WN.REFLEX.EECA

Prog running
About to open database.... WN.REFLEX.EECA
Database opened successfully


ADB_InitEvents -- transStarted

event name is: UPDATE
EventObject::Destroy
ADB_InitEvents -- transStarted

event name is: DELETE
EventObject::Destroy
ADB_InitEvents -- transStarted

event name is: READ
EventObject::Destroy
ADB_InitEvents -- transStarted

event name is: CREATE
EventObject::Destroy
ADB_InitEvents -- transStarted

event name is: START
EventObject::Destroy
ADB_InitEvents -- transStarted

event name is: COMMIT
EventObject::Destroy
ADB_InitEvents -- transStarted

event name is: ABORT
EventObject::DestroytransStarted

INITIZE::>ADBGetRM() trans not Started

NOT Virgin DB ------------
RuleManager name is: RMObject

REFLEX Main Application Menu
============================


  1 .. ATCS Application Menu
  2 .. Rule Management Menu
  3 .. Event Management Menu
  5 .. KnowledgeManager Management Menu
```

```
  X .. Exit

Enter selection 1


Air Traffic Control System Menu
===============================

  1 .. Add a new Aircraft
  2 .. Amend a aircrafts record
  3 .. Delete a aircrafts record
  4 .. Retrieve a aircrafts record
  5 .. List all aircrafts in Database
  7 .. Add/Create a number of aircrafts
  0 .. Raise External Event

  X .. Exit

Enter selection 1

New Aircraft: Please enter aircraft name > BA747

EventDetector::eventRaiseTrans: Raising event from BEFORE TransStart
EventDetector::eventRaiseTrans - Event does NOT affect any rules -
returning! EventDetector::eventRaiseTrans: Raising event from AFTER
TransStart  EventDetector::eventRaiseTrans - Event does NOT affect
any rules - returning! EventDetector::eventRaiseDB-Raising Object
Name : BA747
EventDetector::eventRaiseDB: Raising event from AFTER create Object
EventDetector::eventRaiseDB-Event does NOT affect any rules -
returning!
The new aircraft is: BA747
Enter the current position (Latitude Longitude Height eg 16 03 60)
34 187 14500

the X: 34 Y: 187 Z: 14500

New Aircraft Details

ID                  : BA747
Current Position    : 34 187 14500

Are the above details correct? (Y/N) y

INITIZE::>ADBGetRM() trans not Started

NOT Virgin DB ------------
AObject::putObject()

EventDetector::eventRaiseDB-Raising Object Name : BA747
EventDetector::eventRaiseDB: Raising event from BEFORE putObject
EventDetector::eventRaiseDB-Event does NOT affect any rules -
returning!
AObject::putObject-ActiveRules ... Binding TRUE

AObject::putObject-ActiveRules.... isActive TRUE
AObject::putObject-Back ActiveRules Dictionary put:
AObject::putObject-back from put ExemptRules

about to call Object::putObject(deallocate); :
AObject::putObject-about to call EventDetector-> event Raise
EventDetector::eventRaiseDB-Raising Object Name : BA747
```

```
EventDetector::eventRaiseDB: Raising event from AFTER putObject
EventDetector::eventRaiseDB-Event does NOT affect any rules -
returning!
AObject::putObject-Back from event raise:
Committing aircraft details


EventDetector::eventRaiseTrans: Raising event from BEFORE TransCommit
EventDetector::eventRaiseTrans - Event does NOT affect any rules -
returning! EventDetector::eventRaiseTrans: Raising event from AFTER
TransCommit  EventDetector::eventRaiseTrans - Event does NOT affect
any rules - returning!
Air Traffic Control System Menu
==============================


  1 .. Add a new Aircraft
  2 .. Amend a aircrafts record
  3 .. Delete a aircrafts record
  4 .. Retrieve a aircrafts record
  5 .. List all aircrafts in Database
  7 .. Add/Create a number of aircrafts
  0 .. Raise External Event

  X .. Exit

Enter selection 1

New Aircraft: Please enter aircraft name > BA424
```

```
EventDetector::eventRaiseTrans: Raising event from BEFORE TransStart
EventDetector::eventRaiseTrans - Event does NOT affect any rules -
returning! EventDetector::eventRaiseTrans: Raising event from AFTER
TransStart  EventDetector::eventRaiseTrans - Event does NOT affect
any rules - returning! EventDetector::eventRaiseDB-Raising Object
Name : BA424
EventDetector::eventRaiseDB: Raising event from AFTER create Object
EventDetector::eventRaiseDB-Event does NOT affect any rules -
returning!
The new aircraft is: BA424
Enter the current position (Latitude Longitude Height eg 16 03 60)
37 190 14500

the X: 37 Y: 190 Z: 14500

New Aircraft Details

ID                  : BA424
Current Position    : 37 190 14500

Are the above details correct? (Y/N) y

INITIZE::>ADBGetRM() trans not Started

NOT Virgin DB ------------
AObject::putObject()

EventDetector::eventRaiseDB-Raising Object Name : BA424
EventDetector::eventRaiseDB: Raising event from BEFORE putObject
EventDetector::eventRaiseDB-Event does NOT affect any rules -
returning!
```

```
AObject::putObject-ActiveRules.... isActive TRUE
AObject::putObject-Back ActiveRules Dictionary put:
AObject::putObject-back from put ExemptRules

about to call Object::putObject(deallocate); :
AObject::putObject-about to call EventDetector-> event Raise
EventDetector::eventRaiseDB-Raising Object Name : BA424
EventDetector::eventRaiseDB: Raising event from AFTER putObject
EventDetector::eventRaiseDB-Event does NOT affect any rules -
returning!
AObject::putObject-Back from event raise:
Committing aircraft details


EventDetector::eventRaiseTrans: Raising event from BEFORE TransCommit
EventDetector::eventRaiseTrans - Event does NOT affect any rules -
returning! EventDetector::eventRaiseTrans: Raising event from AFTER
TransCommit  EventDetector::eventRaiseTrans - Event does NOT affect
any rules - returning!
Air Traffic Control System Menu
===============================


  1 .. Add a new Aircraft
  2 .. Amend a aircrafts record
  3 .. Delete a aircrafts record
  4 .. Retrieve a aircrafts record
  5 .. List all aircrafts in Database
  7 .. Add/Create a number of aircrafts
  0 .. Raise External Event

  X .. Exit

Enter selection x


Exiting to Application Menu



REFLEX Main Application Menu
============================


  1 .. ATCS Application Menu
  2 .. Rule Management Menu
  3 .. Event Management Menu
  5 .. KnowledgeManager Management Menu

  X .. Exit

Enter selection 2


Rule Management Menu
====================


  1 .. Add Rule
  2 .. Amend Rule
  3 .. Delete Rule
  4 .. Retrieve Rule
```

```
  5 .. List All Rules

  X .. Main Menu

Enter selection 1

Add Rule: Please enter the rules name > Avoid Aircraft Collision
Please enter description line 1: Triggered when aircraft movements
are detected within the airspace

Please enter description line 2:


Please enter description line 3:


Please enter Event Specification: update aircraft


Please enter Condition String (if OSQL please finish with ';' select
a.Name(), b.Name() from aircraft a, aircraft b where a.Name() =
OBJECT1 and (a.CurX-b.CurX) between -5 and 5 and (a.CurY-b.CurY)
between -5 and 5 and (a.CurZ-b.CurZ) between  -5000 and 5000;

Please enter Action String, either as a SQL query of a function call
i.e. select a.ID() from aircraft a where a.Name() = OBJECT1; please
ensure to put ';' to finish
or    call AlertOperator
call AlertOperator OBJECT1
```

```
INITIZE::>ADBGetRM() trans not Started

NOT Virgin DB ------------
In Rule::RuleManagerAssign

Previous rulemanager is NULL

ruleManager.isNuLL is TRUE, hence does not exist
The new RuleManager's name is RMObject

Entered Rule::putObject... About to put the dictionaries
Rule::putObject now the events
Rule::putObject   now the objects
Rule::putObject   now the clauses array
Rule::putObject   now to set the number...

Rule::putObject   number : RM000001 now to put the rule object
Rule::Rule .. Leaving Constructor

RULE::parseEventSpec    about to create 10 new clause instances
Deleted the pclauses
leaving the Rule::parseEven

** Rule::conditionStr str :select a.Name(), b.Name() from aircraft a,
aircraft b where a.Name() = OBJECT1 and (a.CurX-b.CurX) between -5
and 5 and (a.CurY-b.CurY) between -5 and 5 and (a.CurZ-b.CurZ)
between  -5000 and 5000; ** Rule::conditionStr strlength :197
** Rule::conditionStr Condition : (null)
** Rule::conditionStr Condition  :select a.Name(), b.Name() from
aircraft a, aircraft b where a.Name() = OBJECT1 and (a.CurX-b.CurX)
between -5 and 5 and (a.CurY-b.CurY) between -5 and 5 and
```

```
(a.CurZ-b.CurZ) between  -5000 and 5000; Rule::actionClause
RuleAction::Rule .. Leaving Constructor

Entered RuleAction::putObject... About to put the dictionaries
RuleAction::putObject now the events
Rule::clause, index 0
Rule::ruleClause - Binding is TRUE
Rule::ruleClause - *clauseArray = ((Array*) compClauses.Binding
Rule::ruleClause - about to return cl
Rule::AddEvent
EventObject::AddRule
EventObject::AddRule, fromRule:1
EventObject::putObject
Rule::AddEvent, leaving
Rule::linkEventDependents - After Rule::AddEvent(UPDATE)
Entered Rule::putObject... About to put the dictionaries
Rule::putObject now the events
Rule::putObject   now the objects
Rule::putObject   now the clauses array
Entered RuleAction::putObject... About to put the dictionaries
RuleAction::putObject now the events
Rule::putObject   now to set the number...

Rule::putObject   number : RM000001 now to put the rule object

Rule::putObject eventDestroy is not null

New Rule Details

Name          : Avoid Aircraft Collision          Rule No: RM000001
Description 1: Triggered when aircraft movements are detectedwithin
the airspace              2:
            3:
Event Spec    : UPDATE aircraft
Condition     : select a.Name(), b.Name() from aircraft a, aircraft b
where a.Name() = OBJECT1 and (a.CurX-b.CurX) between -5 and 5 and
(a.CurY-b.CurY) between -5 and 5 and (a.CurZ-b.CurZ) between  -5000
and 5000; Action: call AlertOperator OBJECT1    Immediate   Dependent
Events        : UPDATE

Are the above details correct? (Y/N) y

Entered Rule::putObject... About to put the dictionaries
Rule::putObject now the events
Rule::putObject   now the objects
Rule::putObject   now the clauses array
Entered RuleAction::putObject... About to put the dictionaries
RuleAction::putObject now the events
Rule::putObject   now to set the number...

Rule::putObject   number : RM000001 now to put the rule object
Rule::putObject eventDestroy is not null

Rule Management Menu
====================


  1 .. Add Rule
  2 .. Amend Rule
  3 .. Delete Rule
  4 .. Retrieve Rule
  5 .. List All Rules
```

```
  X .. Main Menu

Enter selection x

Exiting


REFLEX Main Application Menu
============================


  1 .. ATCS Application Menu
  2 .. Rule Management Menu
  3 .. Event Management Menu
  5 .. KnowledgeManager Management Menu

  X .. Exit

Enter selection 1

Air Traffic Control System Menu
===============================


  1 .. Add a new Aircraft
  2 .. Amend a aircrafts record
  3 .. Delete a aircrafts record
  4 .. Retrieve a aircrafts record
  5 .. List all aircrafts in Database
  7 .. Add/Create a number of aircrafts
  0 .. Raise External Event

  X .. Exit

Enter selection 1

New Aircraft: Please enter aircraft name > PK121

EventDetector::eventRaiseTrans: Raising event from BEFORE TransStart
EventDetector::eventRaiseTrans - Event does NOT affect any rules -
returning! EventDetector::eventRaiseTrans: Raising event from AFTER
TransStart  EventDetector::eventRaiseTrans - Event does NOT affect
any rules - returning! EventDetector::eventRaiseDB-Raising Object
Name : PK121
EventDetector::eventRaiseDB: Raising event from AFTER create Object
EventDetector::eventRaiseDB-Event does NOT affect any rules -
returning!
The new aircraft is: PK121
Enter the current position (Latitude Longitude Height eg 16 03 60)
29 183 19000

the X: 29 Y: 183 Z: 19000

New Aircraft Details

ID                  : PK121
Current Position    : 29 183 19000

Are the above details correct? (Y/N) y

INITIZE::>ADBGetRM() trans not Started
```

```
NOT Virgin DB ------------
AObject::putObject()

EventDetector::eventRaiseDB-Raising Object Name : PK121
EventDetector::eventRaiseDB: Raising event from BEFORE putObject Time
is : Mon Jun 26 17:53:28 1995

INITIZE::RETURN new Temp LOG no->ADBGetLogNo() trans not Started
Virgin DB... Initializing TemporalLogManager

Put TLogMan to virgin database after Initializing
INITIZE::>ADBGetRM() trans not Started

NOT Virgin DB ------------
RuleManager::knowledgeScheduler, in RIterator->moreData()...
RuleManager::knowledgeScheduler-Rule Name: Avoid Aircraft Collision
!isDisabled:1 KnowlSel::testEventSpec Rule: Avoid Aircraft Collision,
NoClauses: 1  KnowlSel::testEventSpec -- No previous
PartCompiledEventSpec PartCompEventSpec::OwningRule-Previous rule is
NULL

PartCompEventSpec::OwningRule - owningRule.isNuLL is TRUE, hence does
not exist
PartCompEventSpec::OwningRule - The new Rule's name is Avoid Aircraft
Collision

PartCompEventSpec::OwningRule - completed owningRule.Reset( rule,
this)
In Rule::OwnerOfPCES
Previous is NULL

PCES.isNuLL is TRUE, hence does not exist

Entered Rule::putObject... About to put the dictionaries
Rule::putObject now the events
Rule::putObject   now the objects
Rule::putObject   now the clauses array
Rule::putObject   now to set the number...

Rule::putObject   number : RM000001 now to put the rule object

PartCompEventSpec::putObject  About to store the clauses
PartCompEventSpec::putObject
PartCompEventSpec .. Leaving Constructor

KnowlSel::testEventSpec -- NEW PartCompiledEventSpec object created
PartCompEventSpec::clause, index 0
PartCompEventSpec::ruleCompiledClause - Binding is TRUE
PartCompEventSpec::ruleCompiledClause - *clauseArray = ((Array*)
compiledClauses.Binding PartCompEventSpec::ruleCompiledClause - about
to return cl Rule::clause, index 0
Rule::ruleClause - Binding is TRUE
Rule::ruleClause - *clauseArray = ((Array*) compClauses.Binding
Rule::ruleClause - about to return cl
KnowlSel::testSingleEvent - but what type?
KnowlSel::testSimpleSpec - INTERNAL EVENT
KnowlSel::testEventSpec-after cl=rule->ruleClause(0)- IS SIMPLE EVENT
AObject::putObject-ActiveRules ... Binding TRUE

AObject::putObject-ActiveRules.... isActive TRUE
AObject::putObject-Back ActiveRules Dictionary put:
```

```
AObject::putObject-back from put ExemptRules

about to call Object::putObject(deallocate); :
AObject::putObject-about to call EventDetector-> event Raise
EventDetector::eventRaiseDB-Raising Object Name : PK121
EventDetector::eventRaiseDB: Raising event from AFTER putObject Time
is : Mon Jun 26 17:53:29 1995

INITIZE::RETURN new Temp LOG no->ADBGetLogNo() trans not Started
TLog: NOT Virgin DB
INITIZE::>ADBGetRM() trans not Started

NOT Virgin DB ------------
RuleManager::knowledgeScheduler, in RIterator->moreData()...
RuleManager::knowledgeScheduler-Rule Name: Avoid Aircraft Collision
!isDisabled:1 KnowlSel::testEventSpec Rule: Avoid Aircraft Collision,
NoClauses: 1  PartCompEventSpec::clause, index 0
PartCompEventSpec::ruleCompiledClause - Binding is TRUE
PartCompEventSpec::ruleCompiledClause - *clauseArray = ((Array*)
compiledClauses.Binding PartCompEventSpec::ruleCompiledClause - about
to return cl Rule::clause, index 0
Rule::ruleClause - Binding is TRUE
Rule::ruleClause - *clauseArray = ((Array*) compClauses.Binding
Rule::ruleClause - about to return cl
KnowlSel::testSingleEvent - but what type?
KnowlSel::testSimpleSpec - INTERNAL EVENT
KnowlSel::testSimpleSpec - INTERVALs MATCH
Clause::contextClassTypeName: aircraft
KnowlSel::testSimpleSpec - back from cl->contextClassTypeName(clBuf):
aircraft KnowlSel::testSimpleSpec - TYPES MATCH
KnowlSel::testEventSpec-after cl=rule->ruleClause(0)- IS SIMPLE EVENT
RuleManager::knowledgeScheduler - Rule Avoid Aircraft Collision Event
Specification Satisfied! RuleManager::knowledgeScheduler before
conditionStr
ConditionEvaluator::mapEventParameters - Condition Not NULL Condition
: select a.Name(), b.Name() from aircraft a, aircraft b where
a.Name() = OBJECT1 and (a.CurX-b.CurX) between -5 and 5 and
(a.CurY-b.CurY) between -5 and 5 and (a.CurZ-b.CurZ) between  -5000
and 5000; --> numberOfClauses 1 Rule::clause, index 0
Rule::ruleClause - Binding is TRUE
Rule::ruleClause - *clauseArray = ((Array*) compClauses.Binding
Rule::ruleClause - about to return cl
ConditionEvaluator::mapEventParameters--> Finished ===> About to call
::parseQuery(select a.Name(), b.Name() from aircraft a, aircraft b
where a.Name() =  "PK121"  and (a.CurX-b.CurX) between -5 and 5 and
(a.CurY-b.CurY) between -5 and 5 and (a.CurZ-b.CurZ) between  -5000
and 5000;)
EventDetector::eventRaiseDB-Raising Object Name : BA747
EventDetector::eventRaiseDB: Raising event from AFTER read Object
EventDetector::eventRaiseDB-Event does NOT affect any rules -
returning!
EventDetector::eventRaiseDB-Raising Object Name : BA424
EventDetector::eventRaiseDB: Raising event from AFTER read Object
EventDetector::eventRaiseDB-Event does NOT affect any rules -
returning! "PK121"      "BA747"
"PK121"     "PK121"
Cardinality = 2

RuleManager::knowledgeScheduler Back from Query Evaluation, result: 2
RuleManager::knowledgeScheduler - about to execute Action clausecall
AlertOperator OBJECT1  Rule::clause, index 0
Rule::ruleClause - Binding is TRUE
```

```
Rule::ruleClause - *clauseArray = ((Array*) compClauses.Binding
Rule::ruleClause - about to return cl
ExecutionModule::mapEventParameters--> Finished ===> About to call
::parseQuery(call AlertOperator  "PK121" )
ExecutionModule::executeCommand- CommandType: call    MappedStr: call
AlertOperator  "PK121"   AppObject::executeCommand
AppObject::extractCommand str: AlertOperator  "PK121"
AppObject::extractCommand cp: AlertOperator, str: AlertOperator
"PK121"  AppObject::extractCommand command: AlertOperator
AppObject::syntaxCheck cmdStr: AlertOperator
AppObject::executeCommand - about to switch(cmd)
ATC::AlertOperator *********  Aircraft "PK121"  in Danger Args:
"PK121"  + (null)

AObject::putObject-Back from event raise:
Committing aircraft details


EventDetector::eventRaiseTrans: Raising event from BEFORE TransCommit
EventDetector::eventRaiseTrans - Event does NOT affect any rules -
returning! EventDetector::eventRaiseTrans: Raising event from AFTER
TransCommit  EventDetector::eventRaiseTrans - Event does NOT affect
any rules - returning!
Air Traffic Control System Menu
===============================


  1 .. Add a new Aircraft
  2 .. Amend a aircrafts record
  3 .. Delete a aircrafts record
  4 .. Retrieve a aircrafts record
  5 .. List all aircrafts in Database
  7 .. Add/Create a number of aircrafts
  0 .. Raise External Event

  X .. Exit

Enter selection x


Exiting to Application Menu



REFLEX Main Application Menu
============================


  1 .. ATCS Application Menu
  2 .. Rule Management Menu
  3 .. Event Management Menu
  5 .. KnowledgeManager Management Menu

  X .. Exit

Enter selection 2


Rule Management Menu
====================
```

```
  1 .. Add Rule
  2 .. Amend Rule
  3 .. Delete Rule
  4 .. Retrieve Rule
  5 .. List All Rules

  X .. Main Menu

Enter selection 5

Name          : Avoid Aircraft Collision          Rule No: RM000001
Description 1: Triggered when aircraft movements are detectedwithin
the airspace              2:
            3:
Event Spec    : UPDATE aircraft
Condition     : select a.Name(), b.Name() from aircraft a, aircraft b
where a.Name() = OBJECT1 and (a.CurX-b.CurX) between -5 and 5 and
(a.CurY-b.CurY) between -5 and 5 and (a.CurZ-b.CurZ) between  -5000
and 5000; Action: call AlertOperator OBJECT1    Immediate   Dependent
Events        : UPDATE


Rule Management Menu
====================


  1 .. Add Rule
  2 .. Amend Rule
  3 .. Delete Rule
  4 .. Retrieve Rule
  5 .. List All Rules

  X .. Main Menu

Enter selection x


Exiting



REFLEX Main Application Menu
============================


  1 .. ATCS Application Menu
  2 .. Rule Management Menu
  3 .. Event Management Menu
  5 .. KnowledgeManager Management Menu

  X .. Exit

Enter selection 3


Event Management Menu
=====================

  0 .. Raise Event

  1 .. Add Event
  2 .. Amend Event
```

```
  3 .. Delete Event
  4 .. Retrieve Event
  5 .. List All Events
  7 .. UNAssign a rule from an event

  X .. Main Menu

Enter selection 1

Add Event: Please enter the event name > RadarPulse

Please enter description line 1: Event is raised when aircraft
movement is detected

Please enter description line 2: within its airspace


Please enter description line 3:



New Event Details

Name                  : RadarPulse     Num of Rules: 0
1:    Event is raised when aircraft movement is detected
2:    within its airspace
3:

Are the above details correct? (Y/N) y

INITIZE::>ADBGetRM() trans not Started

NOT Virgin DB ------------
EventObject::putObject
Committing event details



Event Management Menu
=====================

  0 .. Raise Event

  1 .. Add Event
  2 .. Amend Event
  3 .. Delete Event
  4 .. Retrieve Event
  5 .. List All Events
  7 .. UNAssign a rule from an event

  X .. Main Menu

Enter selection x

Exiting Event Management Menu


REFLEX Main Application Menu
============================
```

```
  1 .. ATCS Application Menu
  2 .. Rule Management Menu
  3 .. Event Management Menu
  5 .. KnowledgeManager Management Menu

  X .. Exit

Enter selection 2


Rule Management Menu
====================


  1 .. Add Rule
  2 .. Amend Rule
  3 .. Delete Rule
  4 .. Retrieve Rule
  5 .. List All Rules

  X .. Main Menu

Enter selection 2

Amend Rule: Please enter rule name > Avoid Aircraft Collision
Name          : Avoid Aircraft Collision          Rule No: RM000001
Description 1: Triggered when aircraft movements are detectedwithin
the airspace              2:
            3:
Event Spec    : UPDATE aircraft
Condition     : select a.Name(), b.Name() from aircraft a, aircraft b
where a.Name() = OBJECT1 and (a.CurX-b.CurX) between -5 and 5 and
(a.CurY-b.CurY) between -5 and 5 and (a.CurZ-b.CurZ) between  -5000
and 5000; Action: call AlertOperator OBJECT1    Immediate   Dependent
Events        : UPDATE

Select option      (X)Abort, (Y)Accept and Commit
             Change (E)ESL, (C)Condition, (A)Action >> e

Please enter Event Specification: event RadarPulse or after update
aircraft
RULE::parseEventSpec    about to create 10 new clause instances
Deleted the pclauses
leaving the Rule::parseEven

Name          : Avoid Aircraft Collision          Rule No: RM000001
Description 1: Triggered when aircraft movements are detectedwithin
the airspace              2:
            3:
Event Spec    : EVENT RadarPulse OR AFTER UPDATE aircraft
Condition     : select a.Name(), b.Name() from aircraft a, aircraft b
where a.Name() = OBJECT1 and (a.CurX-b.CurX) between -5 and 5 and
(a.CurY-b.CurY) between -5 and 5 and (a.CurZ-b.CurZ) between  -5000
and 5000; Action: call AlertOperator OBJECT1    Immediate   Dependent
Events        : UPDATE

Select option      (X)Abort, (Y)Accept and Commit
             Change (E)ESL, (C)Condition, (A)Action >> y

Rule::clause, index 0
Rule::ruleClause - Binding is TRUE
Rule::ruleClause - *clauseArray = ((Array*) compClauses.Binding
```

```
Rule::ruleClause - about to return cl
Rule::AddEvent
EventObject::AddRule
EventObject::AddRule, fromRule:1
EventObject::putObject
Rule::AddEvent, leaving
Rule::linkEventDependents - After Rule::AddEvent(RadarPulse)
Rule::clause, index 1
Rule::ruleClause - Binding is TRUE
Rule::ruleClause - *clauseArray = ((Array*) compClauses.Binding
Rule::ruleClause - about to return cl
Rule::AddEvent
Rule::AddEvent, leaving
Rule::linkEventDependents - After Rule::AddEvent(UPDATE)
Entered Rule::putObject... About to put the dictionaries
Rule::putObject now the events
Rule::putObject   now the objects
Rule::putObject   now the clauses array
Entered RuleAction::putObject... About to put the dictionaries
RuleAction::putObject now the events
Rule::putObject   now to set the number...

Rule::putObject   number : RM000001 now to put the rule object

Rule::putObject eventDestroy is not null
Committing Rule details

Rule Management Menu
====================


  1 .. Add Rule
  2 .. Amend Rule
  3 .. Delete Rule
  4 .. Retrieve Rule
  5 .. List All Rules

  X .. Main Menu

Enter selection x


Exiting


REFLEX Main Application Menu
============================


  1 .. ATCS Application Menu
  2 .. Rule Management Menu
  3 .. Event Management Menu
  5 .. KnowledgeManager Management Menu

  X .. Exit

Enter selection 1
```

```
==============================


  1 .. Add a new Aircraft
  2 .. Amend a aircrafts record
  3 .. Delete a aircrafts record
  4 .. Retrieve a aircrafts record
  5 .. List all aircrafts in Database
  7 .. Add/Create a number of aircrafts
  0 .. Raise External Event

  X .. Exit

Enter selection 5

EventDetector::eventRaiseDB-Raising Object Name : BA747
EventDetector::eventRaiseDB: Raising event from AFTER read Object
EventDetector::eventRaiseDB-Event does NOT affect any rules -
returning!
ID    : BA747       Name : BA747       POS   : 34 187 14500
EventDetector::eventRaiseDB-Raising Object Name : BA424
EventDetector::eventRaiseDB: Raising event from AFTER read Object
EventDetector::eventRaiseDB-Event does NOT affect any rules -
returning!
ID    : BA424       Name : BA424       POS   : 37 190 14500
EventDetector::eventRaiseDB-Raising Object Name : PK121
EventDetector::eventRaiseDB: Raising event from AFTER read Object
EventDetector::eventRaiseDB-Event does NOT affect any rules -
returning!
ID    : PK121       Name : PK121       POS   : 29 183 19000

Air Traffic Control System Menu
==============================


  1 .. Add a new Aircraft
  2 .. Amend a aircrafts record
  3 .. Delete a aircrafts record
  4 .. Retrieve a aircrafts record
  5 .. List all aircrafts in Database
  7 .. Add/Create a number of aircrafts
  0 .. Raise External Event

  X .. Exit

Enter selection 2

Amend Aircraft: Please enter aircraft name > PK121

EventDetector::eventRaiseDB-Raising Object Name : PK121
EventDetector::eventRaiseDB: Raising event from AFTER read Object
EventDetector::eventRaiseDB-Event does NOT affect any rules -
returning!
ID                    : PK121
Current Position      : 29 183 19000
Enter the new position (Latitude Longitude Height eg 16 03 60) 33 188
19500

the X: 33 Y: 188 Z: 19500

New Aircraft Details
```

```
ID                  : PK121
Current Position    : 33 188 19500

Are the above details correct? (Y/N) y

EventDetector::eventRaiseTrans: Raising event from BEFORE TransStart
EventDetector::eventRaiseTrans - Event does NOT affect any rules -
returning! EventDetector::eventRaiseTrans: Raising event from AFTER
TransStart  EventDetector::eventRaiseTrans - Event does NOT affect
any rules - returning! INITIZE::>ADBGetRM() trans not Started

NOT Virgin DB ------------
AObject::putObject()

EventDetector::eventRaiseDB-Raising Object Name : PK121
EventDetector::eventRaiseDB: Raising event from BEFORE putObject Time
is : Mon Jun 26 19:49:14 1995

INITIZE::RETURN new Temp LOG no->ADBGetLogNo() trans not Started
TLog: NOT Virgin DB
INITIZE::>ADBGetRM() trans not Started

NOT Virgin DB ------------
RuleManager::knowledgeScheduler, in RIterator->moreData()...
RuleManager::knowledgeScheduler-Rule Name: Avoid Aircraft Collision
!isDisabled:1 KnowlSel::testEventSpec Rule: Avoid Aircraft Collision,
NoClauses: 2  KnowlSel::testEventSpec -- No previous
PartCompiledEventSpec PartCompEventSpec::OwningRule-Previous rule is
NULL

PartCompEventSpec::OwningRule - owningRule.isNuLL is TRUE, hence does
not exist
PartCompEventSpec::OwningRule - The new Rule's name is Avoid Aircraft
Collision

PartCompEventSpec::OwningRule - completed owningRule.Reset( rule,
this)
In Rule::OwnerOfPCES

Previous is NULL

PCES.isNuLL is TRUE, hence does not exist

Entered Rule::putObject... About to put the dictionaries
Rule::putObject now the events
Rule::putObject   now the objects
Rule::putObject   now the clauses array
Rule::putObject   now to set the number...

Rule::putObject   number : RM000001 now to put the rule object

PartCompEventSpec::putObject  About to store the clauses
PartCompEventSpec::putObject
PartCompEventSpec .. Leaving Constructor

KnowlSel::testEventSpec -- NEW PartCompiledEventSpec object created
PartCompEventSpec::clause, index 0
PartCompEventSpec::ruleCompiledClause - Binding is TRUE
PartCompEventSpec::ruleCompiledClause - *clauseArray = ((Array*)
compiledClauses.Binding Index: 0
PartCompEventSpec::ruleCompiledClause - compiledClauses 0 > index 0
PartCompEventSpec::ruleCompiledClause - about to return cl
```

```
KnowlSel::testEventSpec - returned from getCompiledClause(index)
KnowlSel::testEventSpec - no clause create new cl
KnowlSel::testEventSpec - cl not satisfied
Rule::clause, index 0
Rule::ruleClause - Binding is TRUE
Rule::ruleClause - *clauseArray = ((Array*) compClauses.Binding
Rule::ruleClause - about to return cl
KnowlSel::testSingleEvent - but what type?
KnowlSel::testEventSpec - COMPLEX EVENT, clause 0 satisfied
PartCompEventSpec::clause, index 1
PartCompEventSpec::ruleCompiledClause - Binding is TRUE
PartCompEventSpec::ruleCompiledClause - *clauseArray = ((Array*)
compiledClauses.Binding Index: 1
PartCompEventSpec::ruleCompiledClause - compiledClauses 1 > index 1
PartCompEventSpec::ruleCompiledClause - about to return cl
KnowlSel::testEventSpec - returned from getCompiledClause(index)
KnowlSel::testEventSpec - no clause create new cl
KnowlSel::testEventSpec - cl not satisfied
Rule::clause, index 1
Rule::ruleClause - Binding is TRUE
Rule::ruleClause - *clauseArray = ((Array*) compClauses.Binding
Rule::ruleClause - about to return cl
KnowlSel::testSingleEvent - but what type?
KnowlSel::testSimpleSpec - INTERNAL EVENT
KnowlSel::expressionEval - Test RPN :  OR C1 C0  - length: 10
-indexPos 0
At While: OR
KnowlSel:: evalClause: OR at pos 4
KnowlSel:: evalClause Caluse OR is numbered 0
PartCompEventSpec::clause, index 0
PartCompEventSpec::ruleCompiledClause - Binding is TRUE
PartCompEventSpec::ruleCompiledClause - *clauseArray = ((Array*)
compiledClauses.Binding Index: 0
PartCompEventSpec::ruleCompiledClause - compiledClauses 0 > index 0
PartCompEventSpec::ruleCompiledClause - about to return cl
KnowlSel::testEventSpec - returned after expressionEval - Returned :1
KnowlSel::testEventSpec-Complex Event Returned TRUE!  Will return to
RuleManager after delete pces PartCompEventSpec::deleteObject
In Rule::OwnerOfPCES

Previous is ACTIVE

Entered Rule::putObject... About to put the dictionaries
Rule::putObject now the events
Rule::putObject   now the objects
Rule::putObject   now the clauses array
Rule::putObject   now to set the number...

Rule::putObject   number : RM000001 now to put the rule object

Entered Rule::putObject... About to put the dictionaries
Rule::putObject now the events
Rule::putObject   now the objects
Rule::putObject   now the clauses array
Rule::putObject   now to set the number...

Rule::putObject   number : RM000001 now to put the rule object

PartCompEventSpec::deleteObject Deleted the owningRule

PartCompEventSpec::deleteObject Deleted the clause array
PartCompEventSpec::deleteObject completed
```

```
RuleManager::knowledgeScheduler - Rule Avoid Aircraft Collision Event
Specification Satisfied! RuleManager::knowledgeScheduler before
conditionStr
ConditionEvaluator::mapEventParameters - Condition Not NULL Condition
: select a.Name(), b.Name() from aircraft a, aircraft b where
a.Name() = OBJECT1 and (a.CurX-b.CurX) between -5 and 5 and
(a.CurY-b.CurY) between -5 and 5 and (a.CurZ-b.CurZ) between  -5000
and 5000; --> numberOfClauses 2 Rule::clause, index 0
Rule::ruleClause - Binding is TRUE
Rule::ruleClause - *clauseArray = ((Array*) compClauses.Binding
Rule::ruleClause - about to return cl
Rule::clause, index 1
Rule::ruleClause - Binding is TRUE
Rule::ruleClause - *clauseArray = ((Array*) compClauses.Binding
Rule::ruleClause - about to return cl
ConditionEvaluator::mapEventParameters--> Finished ===> About to call
::parseQuery(select a.Name(), b.Name() from aircraft a, aircraft b
where a.Name() =  "PK121"  and (a.CurX-b.CurX) between -5 and 5 and
(a.CurY-b.CurY) between -5 and 5 and (a.CurZ-b.CurZ) between  -5000
and 5000;)
EventDetector::eventRaiseDB-Raising Object Name : BA747
EventDetector::eventRaiseDB: Raising event from AFTER read Object
EventDetector::eventRaiseDB-Event does NOT affect any rules -
returning!
EventDetector::eventRaiseDB-Raising Object Name : BA424
EventDetector::eventRaiseDB: Raising event from AFTER read Object
EventDetector::eventRaiseDB-Event does NOT affect any rules -
returning! "PK121"      "BA747"
"PK121"     "BA424"
"PK121"     "PK121"
Cardinality = 3

RuleManager::knowledgeScheduler Back from Query Evaluation, result: 3
RuleManager::knowledgeScheduler - about to execute Action clausecall
AlertOperator OBJECT1  Rule::clause, index 0
Rule::ruleClause - Binding is TRUE
Rule::ruleClause - *clauseArray = ((Array*) compClauses.Binding
Rule::ruleClause - about to return cl
Rule::clause, index 1
Rule::ruleClause - Binding is TRUE
Rule::ruleClause - *clauseArray = ((Array*) compClauses.Binding
Rule::ruleClause - about to return cl
ExecutionModule::mapEventParameters--> Finished ===> About to call
::parseQuery(call AlertOperator  "PK121" )
ExecutionModule::executeCommand- CommandType: call    MappedStr: call
AlertOperator  "PK121"   AppObject::executeCommand
AppObject::extractCommand str: AlertOperator  "PK121"
AppObject::extractCommand cp: AlertOperator, str: AlertOperator
"PK121"  AppObject::extractCommand command: AlertOperator
AppObject::syntaxCheck cmdStr: AlertOperator
AppObject::executeCommand - about to switch(cmd)
ATC::AlertOperator *********  Aircraft "PK121"  in Danger Args:
"PK121"  + (null)

AObject::putObject-Back from event raise:


Air Traffic Control System Menu
===============================


  1 .. Add a new Aircraft
```

```
  2 .. Amend a aircrafts record
  3 .. Delete a aircrafts record
  4 .. Retrieve a aircrafts record
  5 .. List all aircrafts in Database
  7 .. Add/Create a number of aircrafts
  0 .. Raise External Event

  X .. Exit

Enter selection x


Exiting to Application Menu


REFLEX Main Application Menu
============================


  1 .. ATCS Application Menu
  2 .. Rule Management Menu
  3 .. Event Management Menu
  5 .. KnowledgeManager Management Menu

  X .. Exit

Enter selection x


Exiting


About to shutdown

After shutdown
Completed run
```

2. Student Records System

The following two sample runs show (i.) A run from the Vis user interface which sets up the knowledge base, and (ii.) A text based interrogation of the system, where an external event is raised.

2.1. Vis Interaction

```
vis &

ADB_InitEvents -- transStarted

event name is: UPDATE
EventObject::Destroy
ADB_InitEvents -- transStarted

event name is: DELETE
EventObject::Destroy
ADB_InitEvents -- transStarted
```

```
event name is: READ
EventObject::Destroy
ADB_InitEvents -- transStarted

event name is: CREATE
EventObject::Destroy
ADB_InitEvents -- transStarted

event name is: START
EventObject::Destroy
ADB_InitEvents -- transStarted

event name is: COMMIT
EventObject::Destroy
ADB_InitEvents -- transStarted

event name is: ABORT
EventObject::DestroytransStarted

INITIZE::>ADBGetRM() trans not Started

NOT Virgin DB ------------
RuleManager name is: RMObject
!last_message: DataBase Not Active deleted

The Last Message is now set to: Please Wait...
the new_mess is: Database: WN.REFLEX.EECA Active
!last_message: Please Wait... deleted

The Last Message is now set to: Amend Event
the new_mess is: List all Events
!last_message: Amend Event deleted

The Last Message is now set to: List all Events
the new_mess is: Database: WN.REFLEX.EECA Active

Inside the eventSelect function

the string passed is : RunReport
Rstr is : RunReport

!last_message: List all Events deleted
The Last Message is now set to: Database: WN.REFLEX.EECA Active
the new_mess is: Database: WN.REFLEX.EECA ActiveamendEventDetails() -
now will it amend or not?

EventObject::putObject
EventObject::Destroy
Name : RunReport

...rule put!.....

!last_message: Database: WN.REFLEX.EECA Active deleted

The Last Message is now set to: Database: WN.REFLEX.EECA Active
the new_mess is: Capture New Rule Details
!last_message: Database: WN.REFLEX.EECA Active deleted

The Last Message is now set to: Capture New Rule Details
the new_mess is: Database: WN.REFLEX.EECA Active

Name is not Blank: OnReport
```

```
INITIZE::>ADBGetRM() trans not Started

NOT Virgin DB ------------
In Rule::RuleManagerAssign

Previous rulemanager is NULL

ruleManager.isNuLL is TRUE, hence does not exist

The new RuleManager's name is RMObject

Entered Rule::putObject... About to put the dictionaries
Rule::putObject now the events
Rule::putObject   now the objects
Rule::putObject   now the clauses array
Rule::putObject   now to set the number...

Rule::putObject   number : RM000007 now to put the rule object
Rule::Rule .. Leaving Constructor

RULE::parseEventSpec    about to create 10 new clause instances

PARSER::parse, About to call initize
sizeStack: 100

PARSER::testkeyword: EVENT          word: event
      Application KEYWORD: event
PARSER::testkeyword: RUNREPORT            word: RunReport
      KEYWORD: RunReport
PARSER::parse, About to call ANALYSIS
Stack:
Word: C0
Lastw:

Event Specification: EVENT RunReport

Clause 0:   Text: EVENT RunReport          Keyword: EVENT
Set clauses 0: EVENT RunReport      AObject name: (null)

RPN String
 C0

Delete the clauses
Deleted the pclauses
leaving the Rule::parseEven

TEXTSW_CONTENTS: call WhichReportTypeEND
extPos : 20

** Rule::conditionStr str :call WhichReportType
** Rule::conditionStr strlength :20
** Rule::conditionStr Condition : (null)
** Rule::conditionStr Condition  :call WhichReportType
Capture rule ...  EC value is 0
Rule::clause, index 0
Rule::ruleClause - Binding is TRUE
Rule::ruleClause - *clauseArray = ((Array*) compClauses.Binding
Rule::ruleClause - about to return cl
Rule::AddEvent
EventObject::AddRule
EventObject::AddRule, fromRule:1
```

```
EventObject::putObject
Rule::AddEvent, leaving
Rule::linkEventDependents - After Rule::AddEvent(RunReport)
Entered Rule::putObject... About to put the dictionaries
Rule::putObject now the events
Rule::putObject   now the objects
Rule::putObject   now the clauses array
Rule::putObject   now to set the number...

Rule::putObject   number : RM000007 now to put the rule object

Rule::putObject eventDestroy is not null
Name : OnReport

...rule put!.....

!last_message: Capture New Rule Details deleted

The Last Message is now set to: Database: WN.REFLEX.EECA Active
the new_mess is: Database: WN.REFLEX.EECA Active
!last_message: Database: WN.REFLEX.EECA Active deleted

The Last Message is now set to: Database: WN.REFLEX.EECA Active
the new_mess is: Get Action
ruleActionFlag 0 after Not -1
Rule::actionClause
RuleAction::Rule .. Leaving Constructor

Entered RuleAction::putObject... About to put the dictionaries
RuleAction::putObject now the events
Entered Rule::putObject... About to put the dictionaries
Rule::putObject now the events
Rule::putObject   now the objects
Rule::putObject   now the clauses array
Entered RuleAction::putObject... About to put the dictionaries
RuleAction::putObject now the events
Rule::putObject   now to set the number...

Rule::putObject   number : RM000007 now to put the rule object

Entered Rule::putObject... About to put the dictionaries
Rule::putObject now the events
Rule::putObject   now the objects
Rule::putObject   now the clauses array
Entered RuleAction::putObject... About to put the dictionaries
RuleAction::putObject now the events
Rule::putObject   now to set the number...
Rule::putObject   number : RM000007 now to put the rule object
Committing 'put' operation...
!last_message: Database: WN.REFLEX.EECA Active deleted

The Last Message is now set to: Get Action
the new_mess is: Database: WN.REFLEX.EECA Active
!last_message: Get Action deleted

The Last Message is now set to: Database: WN.REFLEX.EECA Active
the new_mess is: Retrieve Rule
In Retreive rule....

rName :OnReport          rNo :RM000007
This rule is triggered by the RunReport external event and
allows different reports to be executed for different users
```

```
ESL :EVENT RunReport
Cond : call WhichReportType
Prior: 300
In Retreive rule.... 2

In Retreive rule.... after xv_sets
RunReport select Name() from student;
In Retreive rule.... about to show

!last_message: Database: WN.REFLEX.EECA Active deleted

The Last Message is now set to: Database: WN.REFLEX.EECA Active
the new_mess is: Get Action
ruleDepModeSelect - buf: "Fail Action" selected value: 1

ruleActionFlag 1 after Not -2
Rule::actionClause
RuleAction::Rule .. Leaving Constructor

Entered RuleAction::putObject... About to put the dictionaries
RuleAction::putObject now the events
Entered Rule::putObject... About to put the dictionaries
Rule::putObject now the events
Rule::putObject   now the objects
Rule::putObject   now the clauses array
Entered RuleAction::putObject... About to put the dictionaries
RuleAction::putObject now the events
Entered RuleAction::putObject... About to put the dictionaries
RuleAction::putObject now the events
Rule::putObject   now to set the number...

Rule::putObject   number : RM000007 now to put the rule object

Entered Rule::putObject... About to put the dictionaries
Rule::putObject now the events
Rule::putObject   now the objects
Rule::putObject   now the clauses array
Entered RuleAction::putObject... About to put the dictionaries
RuleAction::putObject now the events
Entered RuleAction::putObject... About to put the dictionaries
RuleAction::putObject now the events
Rule::putObject   now to set the number...

Rule::putObject   number : RM000007 now to put the rule object

Committing 'put' operation...
!last_message: Database: WN.REFLEX.EECA Active deleted

The Last Message is now set to: Get Action
the new_mess is: Database: WN.REFLEX.EECA Active
!last_message: Get Action deleted

The Last Message is now set to: Database: WN.REFLEX.EECA Active
the new_mess is: Database: WN.REFLEX.EECA Active
!last_message: Database: WN.REFLEX.EECA Active deleted

The Last Message is now set to: Database: WN.REFLEX.EECA Active
the new_mess is: Retrieve Rule
In Retreive rule....

rName :OnReport          rNo :RM000007
This rule is triggered by the RunReport external event and
```

```
allows different reports to be executed for different users

ESL :EVENT RunReport
Cond : call WhichReportType
Prior: 300
In Retreive rule.... 2

In Retreive rule.... after xv_sets
RunReport select Name() from student; select * from student;
In Retreive rule.... about to show

!last_message: Database: WN.REFLEX.EECA Active deleted

The Last Message is now set to: Retrieve Rule
the new_mess is: Database: WN.REFLEX.EECA Active
!last_message: Retrieve Rule deleted

The Last Message is now set to: Database: WN.REFLEX.EECA Active
the new_mess is: Database not active
>>>nw01@splinter 102 %
```

2.2. Text Based Event Invocation

```
main WN.REFLEX.EECA

Prog running
About to open database.... WN.REFLEX.EECA
Database opened successfully

ADB_InitEvents -- transStarted

event name is: UPDATE
EventObject::Destroy
ADB_InitEvents -- transStarted

event name is: DELETE
EventObject::Destroy
ADB_InitEvents -- transStarted

event name is: READ
EventObject::Destroy
ADB_InitEvents -- transStarted

event name is: CREATE
EventObject::Destroy
ADB_InitEvents -- transStarted

event name is: START
EventObject::Destroy
ADB_InitEvents -- transStarted

event name is: COMMIT
EventObject::Destroy
ADB_InitEvents -- transStarted

event name is: ABORT
EventObject::DestroytransStarted

INITIZE::>ADBGetRM() trans not Started
```

```
NOT Virgin DB -------------
RuleManager name is: RMObjectAbout to call the main_menu fn()

Main Menu
=========


  1 .. Application Menu
  2 .. Rule Menu
  3 .. Event Menu
  5 .. RuleManager Menu

  X .. Exit

Enter selection 3


Event Management Menu
=====================

  0 .. Raise Event

  1 .. Add Event
  2 .. Amend Event
  3 .. Delete Event
  4 .. Retrieve Event
  5 .. List All Events
  7 .. UNAssign a rule from an event

  X .. Main Menu

Enter selection 0

Raise Event: Enter event name > RunReport

Argument List: Please enter any arguments (if any) > computing

ED::eventRaise -  Application event: RunReport
Time is : Sun Jul  2 19:57:56 1995

INITIZE::RETURN new Temp LOG no->ADBGetLogNo() trans not Started
TLog: NOT Virgin DB
INITIZE::>ADBGetRM() trans not Started

NOT Virgin DB -------------
RuleManager::knowledgeScheduler, in RIterator->moreData()...
RuleManager::knowledgeScheduler-Rule Name: OnReport !isDisabled:1
KnowlSel::testEventSpec Rule: OnReport, NoClauses: 1
KnowlSel::testEventSpec -- No previous PartCompiledEventSpec
PartCompEventSpec::OwningRule-Previous rule is NULL

PartCompEventSpec::OwningRule - owningRule.isNuLL is TRUE, hence does
not exist

PartCompEventSpec::OwningRule - The new Rule's name is OnReport

PartCompEventSpec::OwningRule - completed owningRule.Reset( rule,
this)

In Rule::OwnerOfPCES
Previous is NULL
PCES.isNuLL is TRUE, hence does not exist
```

```
Entered Rule::putObject... About to put the dictionaries
Rule::putObject now the events
Rule::putObject   now the objects
Rule::putObject   now the clauses array
Rule::putObject   now to set the number...

Rule::putObject   number : RM000007 now to put the rule object

PartCompEventSpec::putObject  About to store the clauses
PartCompEventSpec::putObject
PartCompEventSpec .. Leaving Constructor

KnowlSel::testEventSpec -- NEW PartCompiledEventSpec object created
PartCompEventSpec::clause, index 0
PartCompEventSpec::ruleCompiledClause - Binding is TRUE
PartCompEventSpec::ruleCompiledClause - *clauseArray = ((Array*)
compiledClauses.Binding
Index: 0
PartCompEventSpec::ruleCompiledClause - compiledClauses 0 > index 0
PartCompEventSpec::ruleCompiledClause - about to return cl
KnowlSel::testEventSpec - returned from getCompiledClause(index)
KnowlSel::testEventSpec - no clause create new cl
KnowlSel::testEventSpec - cl not satisfied
Rule::clause, index 0
Rule::ruleClause - Binding is TRUE
Rule::ruleClause - *clauseArray = ((Array*) compClauses.Binding
Rule::ruleClause - about to return cl
KnowlSel::testSingleEvent - but what type?
KnowlSel::testEventSpec-after cl=rule->ruleClause(0)- IS SIMPLE EVENT
RuleManager::knowledgeScheduler - Rule OnReport Event Specification
Satisfied!
RuleManager::knowledgeScheduler before conditionStr
ConditionEvaluator::mapEventParameters - Condition Not NULL
Condition : call WhichReportType --> numberOfClauses 1
ConditionEvaluator::mapEventParameters--> Finished ===> About to call
::parseQuery(call WhichReportType)

AppObject::executeCommand
AppObject::executeCommand - commandStr: call WhichReportType <->
evArgs: computing
AppObject::extractCommand str: call WhichReportType
AppObject::extractCommand cp: call, str: call WhichReportType
AppObject::extractCommand command: call restOfARgs WhichReportType
AppObject::extractCommand call cp: WhichReportType, str: call
WhichReportType
AppObject::extractCommand call command: WhichReportType
AppObject::executeCommand - cmdStr: WhichReportType commandStr: call
WhichReportType Args: WhichReportType
AppObject::syntaxCheck commandStr: WhichReportType
AppObject::executeCommand - cmdStr: WhichReportType cmdNo: 3
AppObject::executeCommand - about to switch(call WhichReportType) ->
evArgs: computing
SRS::WhichReportType    External Condition test, test for Computing
School
SRS::WhichReportType    Args: computing
ConditionEvaluator::returned from executeCommand: 1
RuleManager::knowledgeScheduler Back from Query Evaluation, result: 1
RuleManager::knowledgeScheduler - about to execute Action
clauseselect Name() from student;
ExecutionModule::mapEventParameters--> Finished ===> About to call
::parseQuery(select Name() from student;)
```

```
ExecutionModule::executeCommand- CommandType: select  MappedStr:
select Name() from student;
EventDetector::eventRaiseDB-Raising Object Name : (null)
EventDetector::eventRaiseDB: Raising event from AFTER read Object
EventDetector::eventRaiseDB-Event does NOT affect any rules -
returning!
"Waseem"
Cardinality = 1
Raised


Event Management Menu
=====================

  0 .. Raise Event

  1 .. Add Event
  2 .. Amend Event
  3 .. Delete Event
  4 .. Retrieve Event
  5 .. List All Events
  7 .. UNAssign a rule from an event

  X .. Main Menu

Enter selection 0

Raise Event: Enter event name > RunReport

Argument List: Please enter any arguments (if any) > Mathematics

ED::eventRaise -  Application event: RunReport
Time is : Sun Jul  2 19:58:18 1995

INITIZE::RETURN new Temp LOG no->ADBGetLogNo() trans not Started
TLog: NOT Virgin DB
INITIZE::>ADBGetRM() trans not Started

NOT Virgin DB ------------
RuleManager::knowledgeScheduler, in RIterator->moreData()...
RuleManager::knowledgeScheduler-Rule Name: OnReport !isDisabled:1
KnowlSel::testEventSpec Rule: OnReport, NoClauses: 1
PartCompEventSpec::clause, index 0
PartCompEventSpec::ruleCompiledClause - Binding is TRUE
PartCompEventSpec::ruleCompiledClause - *clauseArray = ((Array*)
compiledClauses.Binding
Index: 0
PartCompEventSpec::ruleCompiledClause - compiledClauses 0 > index 0
PartCompEventSpec::ruleCompiledClause - about to return cl
KnowlSel::testEventSpec - returned from getCompiledClause(index)
KnowlSel::testEventSpec - no clause create new cl
KnowlSel::testEventSpec - cl not satisfied
Rule::clause, index 0
Rule::ruleClause - Binding is TRUE
Rule::ruleClause - *clauseArray = ((Array*) compClauses.Binding
Rule::ruleClause - about to return cl
KnowlSel::testSingleEvent - but what type?
KnowlSel::testEventSpec-after cl=rule->ruleClause(0)- IS SIMPLE EVENT
RuleManager::knowledgeScheduler - Rule OnReport Event Specification
Satisfied!
RuleManager::knowledgeScheduler before conditionStr
ConditionEvaluator::mapEventParameters - Condition Not NULL
```

```
Condition : call WhichReportType --> numberOfClauses 1
ConditionEvaluator::mapEventParameters--> Finished ===> About to call
::parseQuery(call WhichReportType)

AppObject::executeCommand
AppObject::executeCommand - commandStr: call WhichReportType <->
evArgs: Mathematics
AppObject::extractCommand str: call WhichReportType
AppObject::extractCommand cp: call, str: call WhichReportType
AppObject::extractCommand command: call restOfARgs WhichReportType
AppObject::extractCommand call cp: WhichReportType, str: call
WhichReportType
AppObject::extractCommand call command: WhichReportType
AppObject::executeCommand - cmdStr: WhichReportType commandStr: call
WhichReportType Args: WhichReportType
AppObject::syntaxCheck commandStr: WhichReportType
AppObject::executeCommand - cmdStr: WhichReportType cmdNo: 3
AppObject::executeCommand - about to switch(call WhichReportType) ->
evArgs: Mathematics
SRS::WhichReportType    External Condition test, test for Computing
School
SRS::WhichReportType    Args: Mathematics
External Condition Fail! Non Computing School
ConditionEvaluator::returned from executeCommand: 0
RuleManager::knowledgeScheduler Back from Query Evaluation, result: 0
RuleManager::knowledgeScheduler - about to execute Action
clauseExecutionModule::executeCommand - FailAction! requestedselect *
from student;
ExecutionModule::mapEventParameters--> Finished ===> About to call
::parseQuery(select * from student;)

ExecutionModule::executeCommand- CommandType: select  MappedStr:
select * from student;
EventDetector::eventRaiseDB-Raising Object Name : (null)
EventDetector::eventRaiseDB: Raising event from AFTER read Object
EventDetector::eventRaiseDB-Event does NOT affect any rules -
returning!
"37133"     #1Dictionary      #2Dictionary      "Waseem"    77    16
      3     65    (charPtr*)0xa47d4 "(null)"    #3Dictionary
#4Dictionary      1342028904  (void*)0x8e0c0    634412
Cardinality = 1
Raised


Event Management Menu
=====================

  0 .. Raise Event

  1 .. Add Event
  2 .. Amend Event
  3 .. Delete Event
  4 .. Retrieve Event
  5 .. List All Events
  7 .. UNAssign a rule from an event

  X .. Main Menu

Enter selection x

Exiting Event Management Menu
```

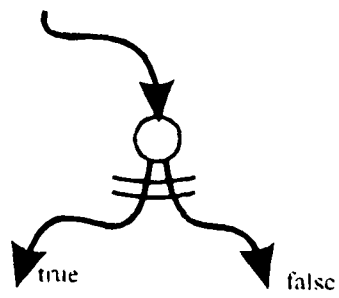
```
Main Menu
=========


  1 .. Application Menu
  2 .. Rule Menu
  3 .. Event Menu
  5 .. RuleManager Menu

  X .. Exit

Enter selection x

Exiting

After main_menu about to call transcommit()
After transcommit()about to call shutdown
After shutdown
After OC_close
23.0u 18.8s 2:40 26% 0+1016k 687+436io 1353pf+0w
>>>nw01@splinter 102 %
```

APPENDIX C

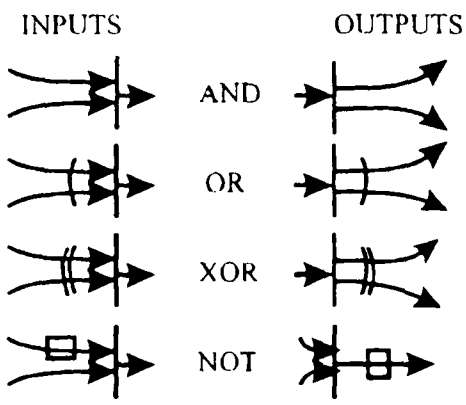
REFLEX Petri Nets

The design for the REFLEX model was tested formally using Petri-Net [Patterson 81] theory. This was considered a good approach for ensuring semantic correctness, and hence was envisaged as being a major constitent of this thesis. However, this goal was down graded after it became evident that other research teams, for instance Navathe, Tanaka and Chakravarthy [Navathe 92], were also adopting the same approach, and had published on it previously. The work presented here in this appendix is sufficiently different to the work of others since the REFLEX model itself was modeled using petri-nets and not just event specifications such as those reported by Gatziu and Dittrich [Gatziu 93].

During the process of modelling REFLEX's logical parts, it was discovered that although petri-net theory was designed for the modelling of complex concurrent systems, such as REFLEX, it did not provide support for logical statements. To address this situation it was found necessary to augment petri-net theory by adding the conditional testing of a place, figure C.1 (a), similar to the approaches by Jensen [Jensen 90] and Baer, Bovet and Estrin [Baer 70]. This can be used singularly or



(a) Conditional place



(b) Logical constructs

Figure C.1 Petri-net extensions

combined to perform additional logic.

To provide further logical functionality the constructs in figure C.1.(b), have been used in this research during the design of the REFLEX active database model.

The following petri-nets were constructed during the early prototypes. The first petri-net shows the Event Manager (EM) and Knowledge Management Kernel (KMK) dialogue, figure C.2.

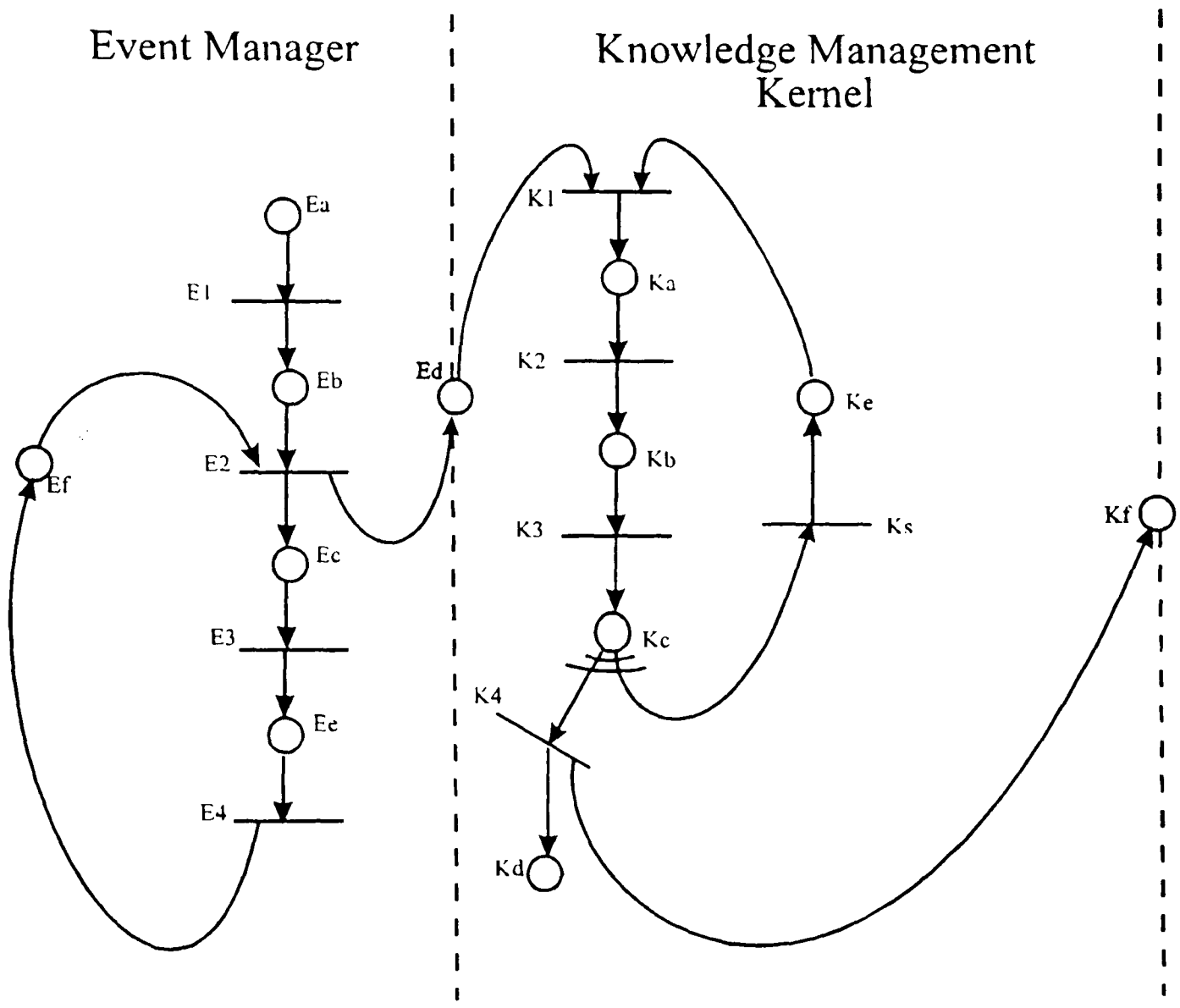


Figure C.2 Petri-net: Event Manager/ Knowledge Management Kernel

The narrative for the above perti-net and others can be found on the following table. After the table are the remainder of the logical petri nets arranged into two graphs, figure C3 and C4, which togther describe the early REFLEX systems.

Conditions		Events	
Ea	EM waiting for an event	E1	EM Event detected
Eb	EM waiting for event to be logged	E2	EM starts to log event
Ec	EM logs the event	E3	EM finishes logging event
Ed	EM informs KMK	E4	EM event logged
Ee	EM event has been logged		
Ef	EM waiting for new event		
Ka	KMK waiting to evaluate event	K1	KMK new event noted
Kb	KMK evaluating event in context	K2	KMK start to evaluate event in context
Kc	KMK does event affect any rule	K3	KMK finish evaluting event in context
Kd	KMK log event in context	K4	KMK event in context
Ke	KMK waiting idle for new event	K5	KMK event not in context
Kf	KMK requests KSM to test event specification	K6	KMK notified rule in context
Kg	KMK wainting to inform CE of valid rules, subject to coupling mode	K7	KMK request CE test condition
Kh	KMK inform CE rule available		

Sa	KSM ready to retrieve rules	S1	KSM event notified
Sb	KSM retrieving rules	S2	KSM start to retrieve events
Sc	KSM rules retrieved	S3	KSM finish retrieving events
Sd	KSM test is event primitive	S4	KSM start to evaluate event
Se	KSM primitive handler waiting idle	S5	KSM rule event specification primitive
Sf	KSM inform KMK primitive	S6	KSM rule event specification complex
Sg	KSM ready to evaluate complex	S7	KSM start evaluating rule event specification
Sh	KSM evaluate complex event and check temporal log	S8	KSM finish valuation
Si	KSM test complex event satisfied	S9	KSM event specification satisfied
Sk	KSM inform KMK complex satisfied	S10	KSM event specification not satisfied
Sl	KSM ready to add to temporal log	S11	KSM start to log part specified event specification
Sm	place on temporal log	S12	KSM finish logging
Ca	CM waiting to evaluate condition	C1	CM rule received
Cb	CM test query type	C2	CM test which language
Cc	CM translate OSQL	C3	CM language OSQL
Cd	CM translate proprietary	C4	CM Language proprietary
Ce	CM evaluate condition	C5	CM start evaluation
Cf	CM condition satisfied	C6	CM finish evaluation
Cg	CM inform KMK condition satisfied	C7	CM condition clause satisfied
		C8	CM condition not satisfied

Aa	ES test action type	A1	ES rule received
Ab	ES waiting to process OSQL	A2	ES action language is OSQL
Ac	ES processing query	A3	ES clause is call to object handler
		A4	ES start action processing
		A7	ES finish action processing

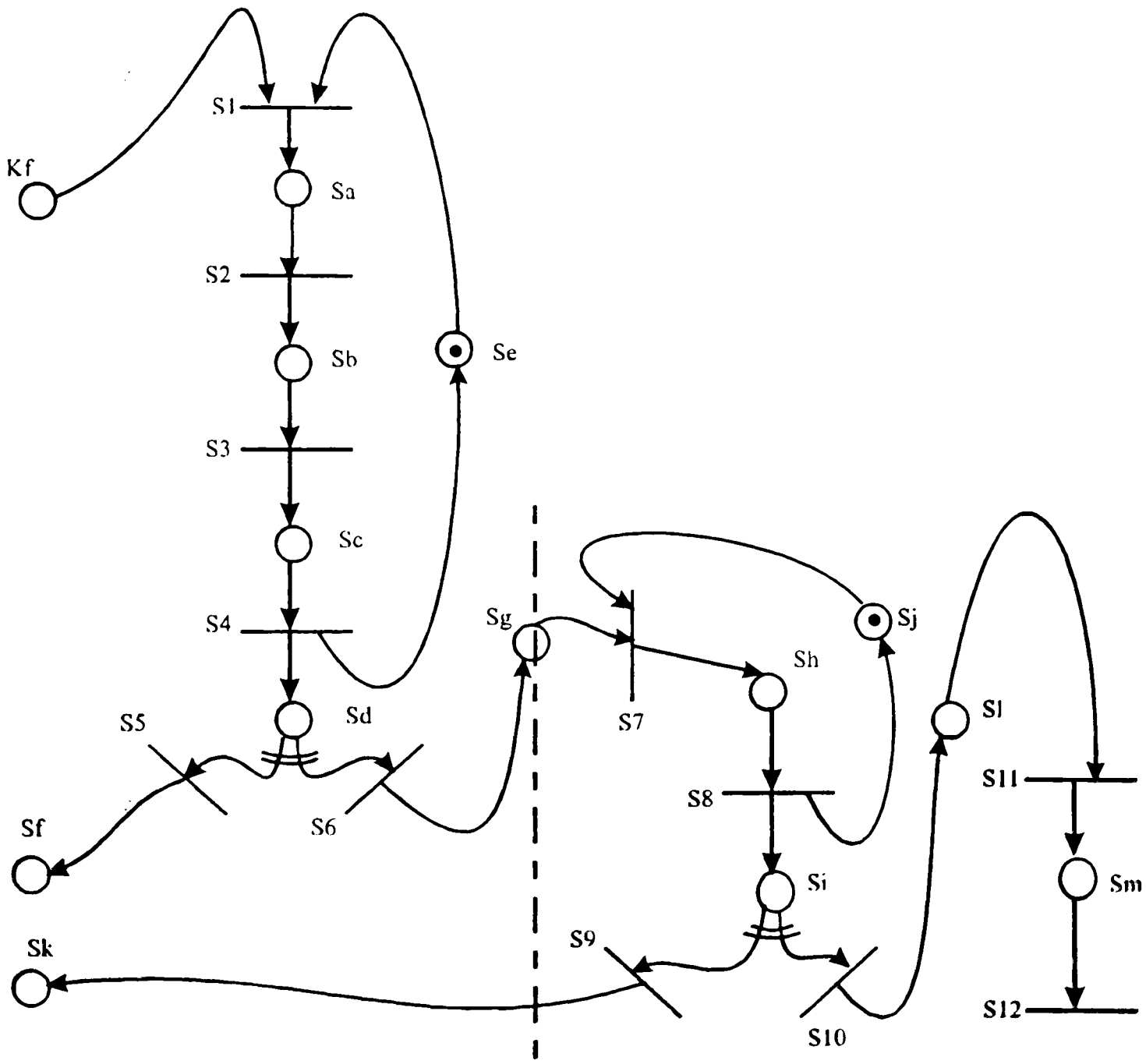


Figure C.3 Petri-net:Knowledge Selection Module

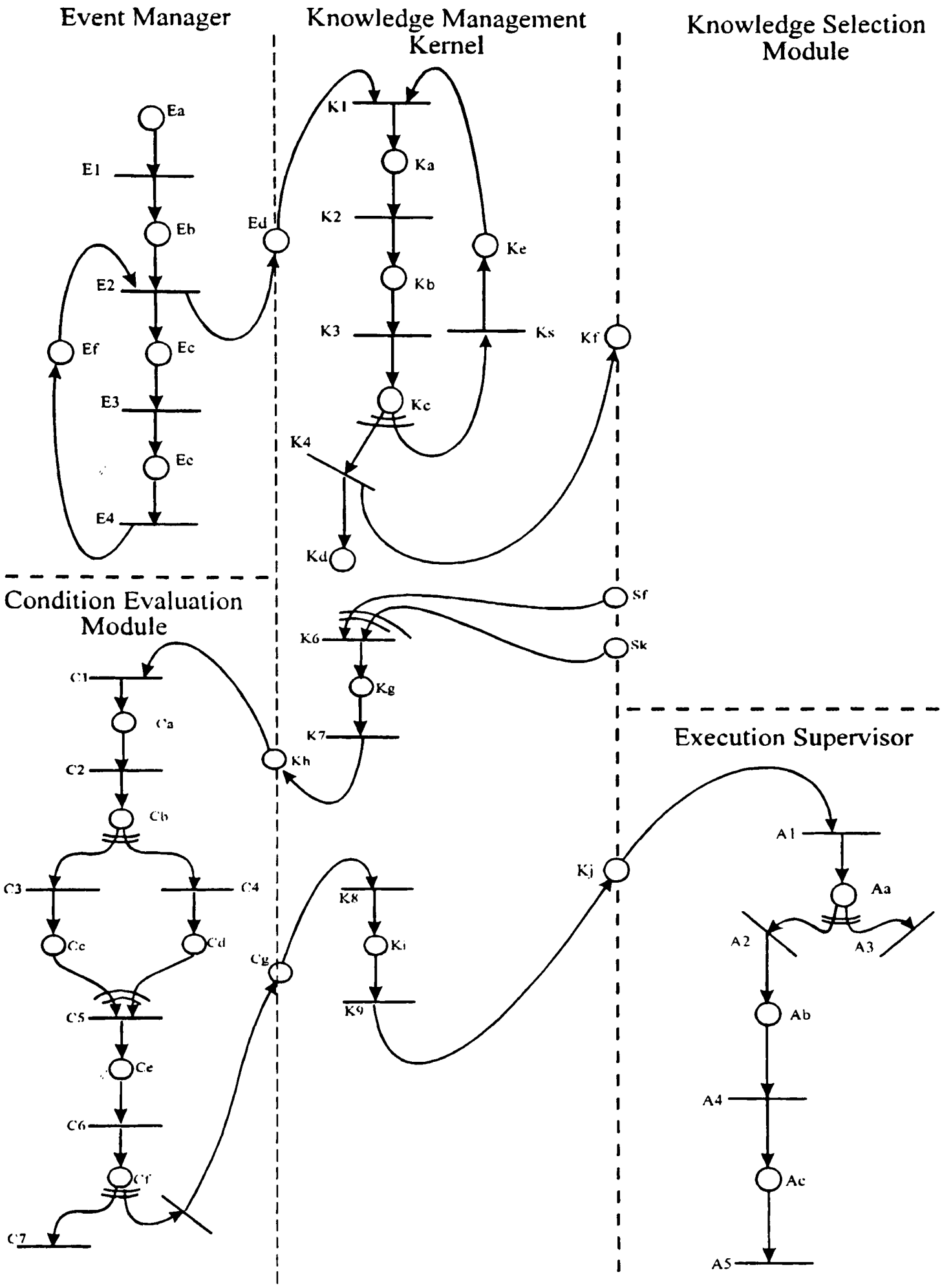
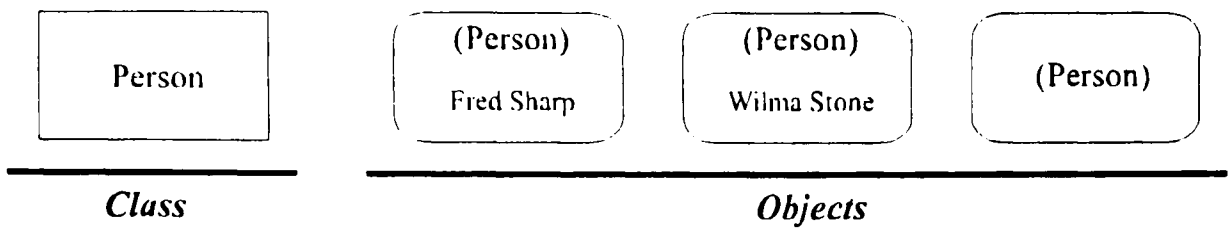


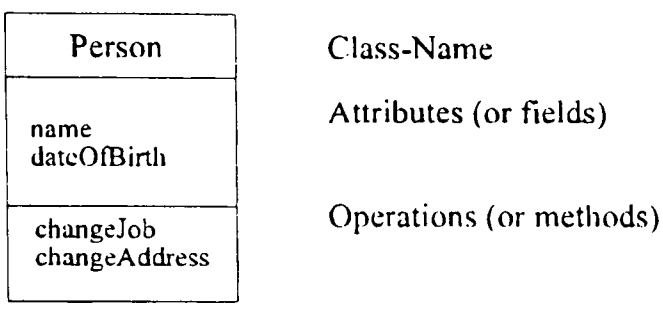
Figure C.4 Petri-net:Major REFLEX Systems

APPENDIX D

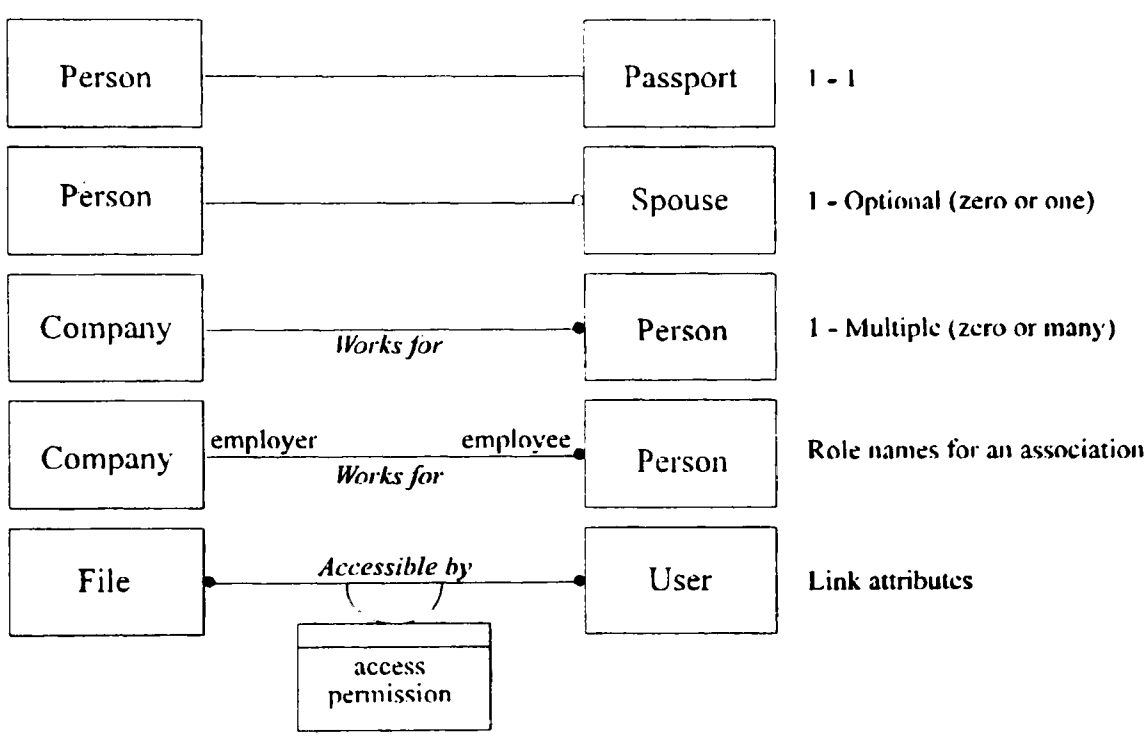
OMT Graphical Notation

A summary of the relevant OMT graphical notaion is presented below. For further details please refer to [Rumbaugh 91].

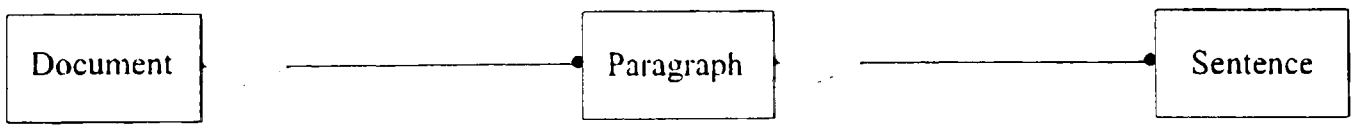




Association (or relationship)



Aggregation



Generalization

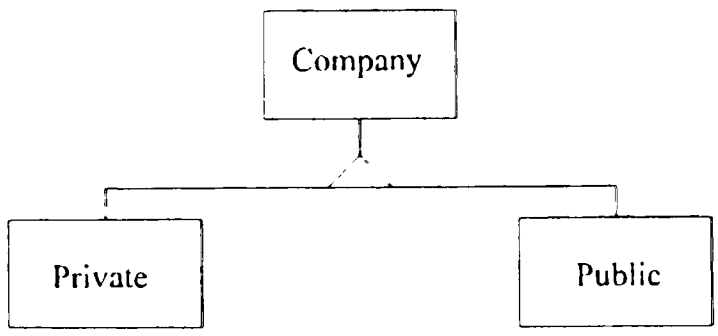


Figure D.1 **OMT Graphical Notation**