Design, Implementation and Evaluation of an Application following the Deductive Approach

Ilias Petrounias and Pericles Loucopoulos
Information Systems Group
Department of Computation
UMIST
P.O. Box 88
Manchester M60 1QD
e-mail:{ilias, pl}@sna.co.umist.ac.uk

Abstract

This paper presents the process and the results of designing and implementing an application using the deductive approach. The application concerns a safety critical system and for modelling the application domain two conceptual models were used: the Entity Relationship Time (ERT) for the structural part and the Conceptual Rule Language (CRL) for constraints, derivation and event-action rules. The mapping from the conceptual level to the deductive DBMS platform, namely MegaLog, is described with some sample results. An overview of the system, along with a simple user interface for creating different application scenarios are presented and are followed by some statistic results of a real life case study and an evaluation of the proposed approach.

1. Introduction

Recent years have witnessed the recognition that applications have become more complex and therefore application programs are needed to be larger and more intelligent and, therefore, more difficult to maintain. The relational technology has been extended as much as possible since it has become a standard in database applications, but now more deductive capabilities are needed to be offered by database systems. Research in the area of deductive systems has grown during the last few years ([Gallaire et al, 1984], [Lloyd and Topor, 1985], [Lloyd and Topor, 1986],[Minker, 1988], [Winslett, 1990]), although, unfortunately it has not yet resulted in many commercial products and the available ones have not yet really been tested.

This paper attempts to summarise some of the results of the use of a deductive system in designing and implementing a safety critical application. Section 2 includes a description of such systems and of the reasons that created the need for deductive capabilities in database systems. Section 3 describes the case study which is the development of a safety critical system, more specifically the handling of a possible attack against a ship, while section 4 describes the conceptual modelling formalisms that were used to model the application domain. Section 5 presents the deductive platform that was used (MegaLog). Section 6 presents the algorithms from mapping the conceptual schemas onto MegaLog and presents some sample results for the structural part. Section 7 presents some of the design and implementation considerations, a user interface for the system for creating different scenarios and provides some statistical results from a real life case study, presents the advantages and disadvantages of the approach that was followed and some problems encountered in the use of MegaLog. Finally section 8 draws some conclusions.

2. The Need for Deductive Capabilities in Database Systems

Relational database systems have been used extensively and successfully in many application areas. Application programs using traditional and fourth generation languages have increased in size and complexity. In order to facilitate complex application development, to avoid code redundancies and multiple developments of similar programs with closed functionalities and to support reasoning capabilities on symbolic data, it is often desirable and beneficial to *integrate* into the database itself a definition of common knowledge shared by different users [Gardarin and Valduriez, 1989].

As a result a new trend emerged within the database research community: database systems that are based on logic. Research on the relationship between the database theory and logic goes back to the 1970s ([Gallaire and Minker, 1978], [Gallaire et al, 1980]). The principal stimulus

for the recent expansion of interest, however, was a paper by Reiter in 1984 [Reiter, 1984], in which it was argued that 'a proof theoretic view of databases was possible, and in fact, preferable than the model theoretic one in certain aspects.

In this alternative view the database is regarded as a set of 'ground' axioms that correspond to tuples in base relations and a set of 'deductive' axioms from which new facts may be derived from facts that have already explicitly introduced [Date, 1990]. The execution of a query is then viewed as a proof that a specified formula is a theorem, a logical consequence of the before mentioned axioms.

Traditional database systems manage the data and meta-data which comprise the *extensional* knowledge that is embedded in facts and instances. The *intentional* knowledge is identified as the knowledge beyond the factual content of the database and this kind of knowledge can be fully specified in the form of rules in a rule base *before* the database is established [Elmasri and Navathe, 1989]. A deductive database system then integrates data manipulation functions and deductive functions into a single system to support *derived* knowledge, in which extensional and intentional knowledge are mixed in order to define derived relations which are often a generalisation of the view concept [Gardarin & Valduriez, 1989].

The objective of such systems is firstly the integration of as much knowledge as possible in the intentional database so that application programs remain efficient, small and easy to write and to maintain, secondly the management in a consistent and efficient way of the rules and data and finally the provision of retrieval and manipulation functions to query and update both the extensional and intentional relations.

3. The Case Study

OWN_SHIP is located in an area with certain coordinates and is travelling with some speed following a particular course. Its location in this area is given by its longitude and latitude and the area in which it is located has some environmental data at the time of interest. The ship's engines have certain capabilities (i.e. power, propulsion etc.). The ship has a number of weapon systems to defend itself against a possible attack each one with certain capabilities and characteristics (i.e. guns, missiles), information providers (i.e. radars) to detect such an attack, logistic information (i.e. status of available fuel) and data concerning possible damages in OWN_SHIP's systems.

At some point in time OWN_SHIP's information providers locate one or more tracks which have certain characteristics (certain type, travelling speed etc.). A system located on the ship will have -based on the information provided- to decide whether or not the detected tracks should be considered as threats to the ship and if they are to formulate a plan in order to prepare a defend against them. In doing so it will have to take into account that the available defence resources might be limited due to possible damages in weapon systems or because the number of threats is too great to defend against or even because some weapons are not usable in the particular situation. Therefore, the system will have to assign priorities to the tracks -depending on their type, distance from OWN_SHIP etc.- in order to deal firstly and be able to use the best available weapons against those that are considered to be the most dangerous ones. Additionally, and because different information providers might give slightly different information about the same tracks, the system after following each track's history must be able to distinguish between 'real' and duplicate tracks. After distinguishing between them the system will only be interested in the 'real' ones.

4. Conceptual Models

The aim at the conceptual level is the development models that will be rich enough to capture and represent knowledge about policy and its rules. Moreover, models should support concepts in a way that communication among the different actors involved in the process of systems development (i.e., users, analysts, designers) can be made easy so as to facilitate the task of maintaining the specification from the initial stage of capturing the user requirements to producing the final running system. In general, three major categories of information about the

world can be identified:

- *Structural knowledge*. Emphasis has been put on the conceptual taxonomies to which the objects of the domain belong. A model has been defined that aims at capturing these objects together with their taxonomies in an efficient and user-friendly manner.

- *Dynamic knowledge*. Emphasis is put on the various functions (processes) of the domain, which change the state of the objects. This aspect, however, can be represented declaratively in the form of rules.

- *Declarative knowledge*. It is concerned with the actual rules that govern both the structural and the dynamic knowledge. Rules should be the central building blocks of any paradigm and a model must be defined which captures such rules and refers to the previous kinds of knowledge.

4.1 The Entity-Relationship-Time Model (ERT)

The ERT model uses as its basic structure the Entity-Relationship approach in order to preserve the well known advantages of this approach namely, graphical display, increased readability and wide acceptance by practitioners. The basic mechanism differs from the original Entity Relationship model [Chen, 1976] in that it regards any association between objects in the unified form of a relationship. Thus, the conceptually unnecessary distinction between attributeships and relationships [Kent, 1979; Nijssen, 1988] is avoided. On this basis, the ERT model offers a number of features which are considered to be necessary for the modelling of complex database applications. Specifically, it accommodates the explicit modelling of time, taxonomic hierarchies and complex objects. The different aspects of data abstraction that are dealt with in the ERT model are: classification, generalisation, aggregation and grouping.

The graphical notation for the different concepts of ERT is shown in figure 1.

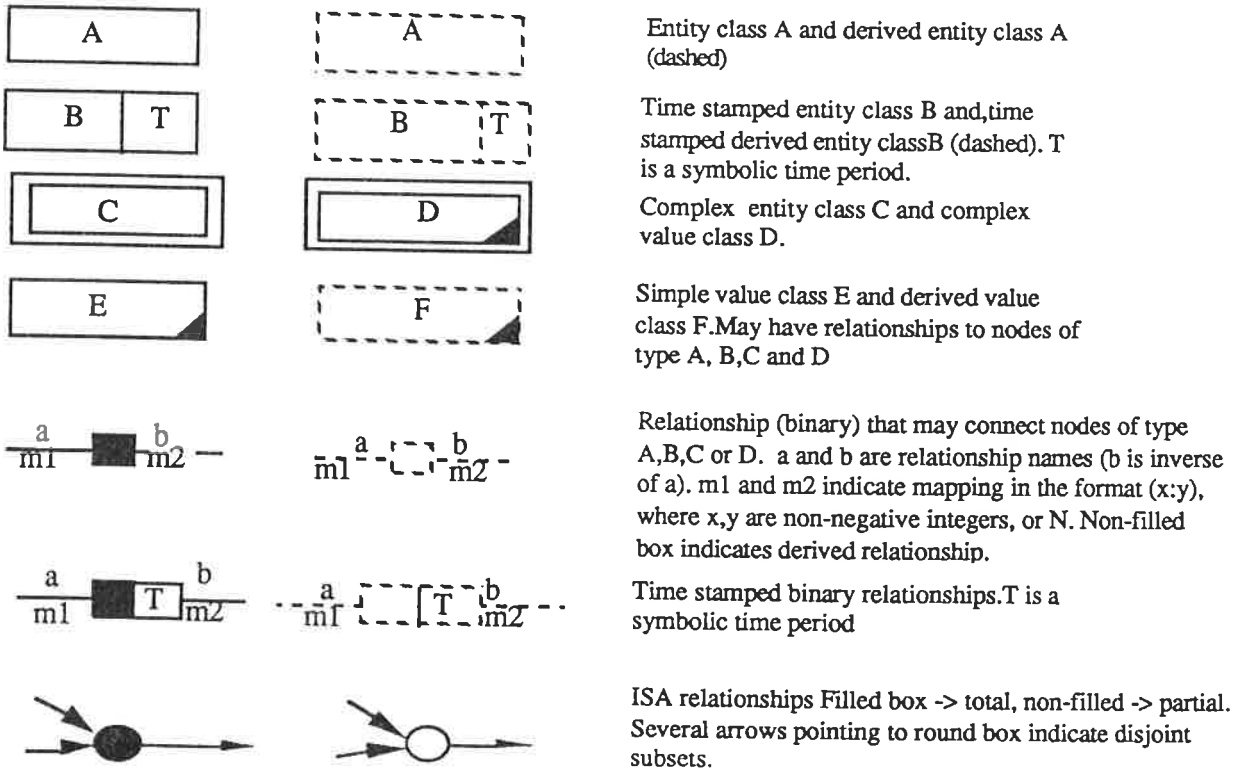

Figure 1: The ERT graphical notation

The need for modelling time explicitly is that, for many applications when an information item becomes outdated, it need not be forgotten. The lack of temporal support raises serious problems in many cases. For example, conventional DBMS cannot support historical queries

about past status, let alone trend analysis which is essential for applications such as Decision Support Systems (DSS). The need to handle time more comprehensively surfaced in the early 70's in the area of medical information systems where a patient's history is particularly important [Wiederhold et al, 1975]. Since these early days there has been a large amount of research in the nature of time in computer-based information systems and the handling of the temporal aspects of data [Ariav and Clifford, 1984; Ahn & Snodgrass, 1988; Ben-Zvi, 1982; Lum et al, 1984; Dadam et al, 1984]. Research interest in the time modelling area has increased dramatically over the past decade as shown by published bibliographies [McKenzie, 1986] and comparative studies [Theodoulidis and Loucopoulos, 1991].

The most primitive concept in ERT is that of a *class* which is defined as a collection of individual objects that have common properties i.e., that are of the same type. In an ERT schema only classes of objects are specified. In addition, every relationship is viewed as a named set of two (entity or value, role) pairs where each role expresses the way that a specific entity or value is involved in a relationship. Time is introduced in ERT as a distinguished class called time period class. More specifically, each time varying entity class and each time varying relationship class is timestamped with a time period class. The term time varying refers to pieces of information that the modeller wants to keep track of their evolution i.e., to keep their history and consequently, to be able to reason about them. For example, for each entity class, a time period might be associated which represents the period of time during which an entity is modelled. This is referred to as the *existence period* of an entity. The same argument applies also to relationships i.e., each time varying relationship might be associated with a time period which represents the period during which the relationship is valid. This is referred to as the *validity period* of a relationship.

A distinction is made between different variations of ISA hierarchies. These are based on two constraints that are usually included in any ISA semantics namely, the *partial/total* ISA constraint and the *disjoint/overlapping* ISA constraint [Theodoulidis et al, 1990] Based on the above constraints the four kinds of ISA relationships are supported namely, Partial Disjoint ISA, Total Disjoint ISA, Partial Overlapping ISA and Total Overlapping ISA.

In ERT, *physical part hierarchies* as well as *logical part hierarchies* are accommodated [Kim et al, 1987; Lorie & Plouffe, 1983; Rabitti et al, 1988]. To achieve this, four different kinds of IS_PART_OF relationships are defined according to two constraints, namely the *dependency* and *exclusiveness* constraints. The dependency constraint states that when a complex object ceases to exist, all its components also cease to exist (dependent composite reference) and the exclusiveness constraint states that a component object can be part of at most one complex object (exclusive composite reference). In ERT, *dependent exclusive, independent exclusive, dependent shared and independent shared IS_PART_OF* variations are accommodated [Theodoulidis et al, 1991a].

The ERT schema for the case study described in the previous section is shown in Figure 2.

4.2 The Conceptual Rule Language

The Conceptual Rule Language (CRL) provides the means for controlling the behaviour of a domain in terms of rules. The role of the CRL is twofold. Firstly, it is concerned with constraints placed upon the elements of ERT and with the derivation of new information based on existing information. Secondly, it is concerned with the eligibility for the firing of actions (performed on the ERT objects) and constraints placed on their order of execution.

In order to provide more assistance in the rule elicitation process, different types of CRL rules have been distinguished. These rule types are the following:

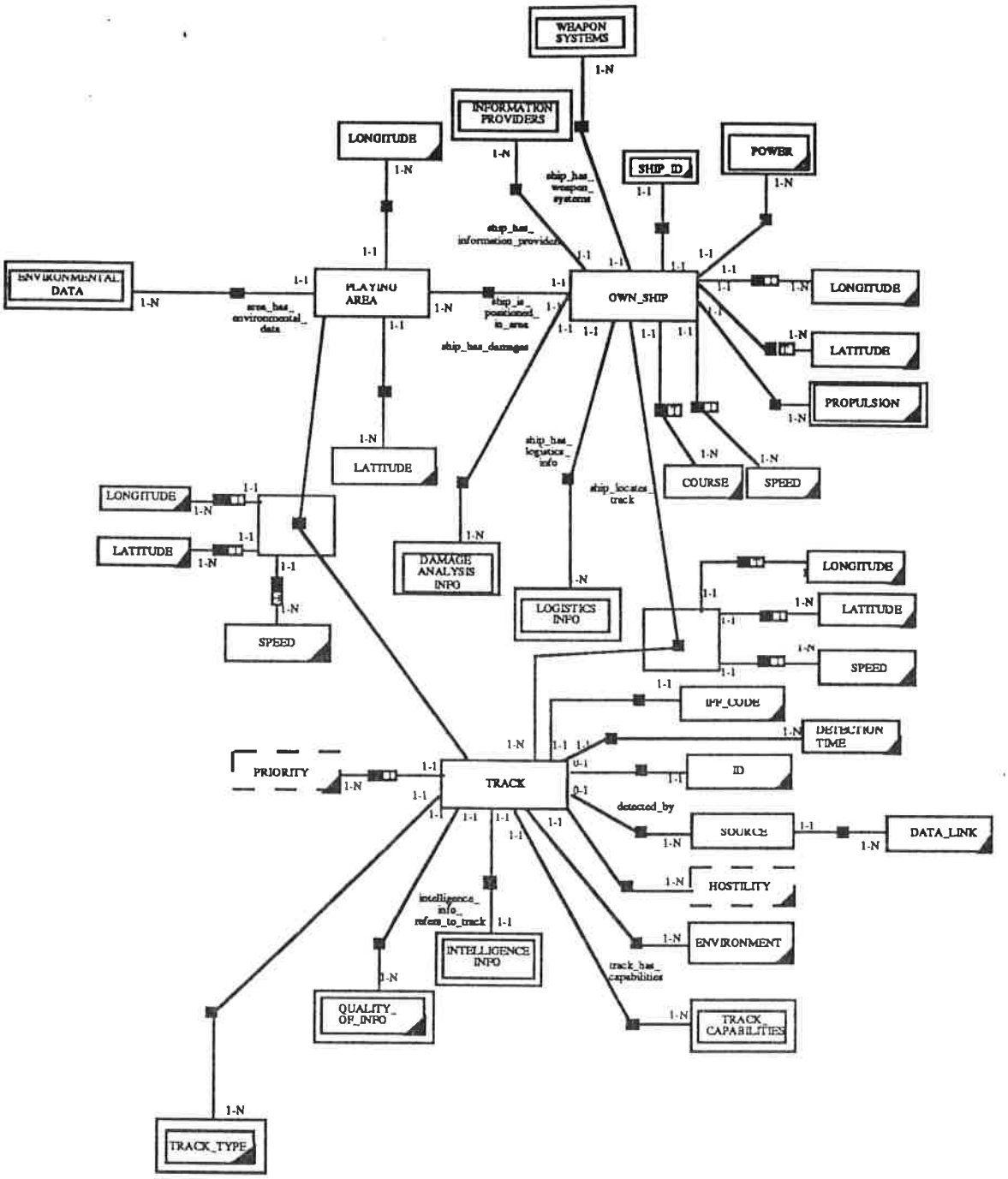

Figure 2: The ERT schema of the application domain

- *Constraint rules* which are concerned with the integrity of the ERT components. More specifically, the constraint rules express restrictions on the ERT components by constraining individual ERT states and state transitions, where a state is defined as the extension of the database at any clock tick.

- *Derivation rules* which are expressions that define the derived components of the ERT model in terms of other ERT components. Derivation rules are introduced as a means of capturing structural domain knowledge that need not be stored and that its value can be derived dynamically using existing or other derived information. For each derived ERT component there should exist one and only one CRL derivation rule.

- *Event-action rules* which are concerned with the invocation of actions. In particular, action rules express the conditions under which the actions must be taken, i.e., a set of

triggering conditions and/or a set of preconditions that must be satisfied prior to their execution.

The definition of the CRL allows us two important benefits. Firstly, we may express the external rules (that the user views) as manipulating data in the ERT model, but have our executable rules manipulate data in the database model, and thus be more efficient to execute. Secondly, we may heavily sugar the syntax of the CRL to give a semi-natural language flavour, whilst leaving the internal rule language in a more concise form that programmers would desire.

4.2.1 Referencing the ERT Model

The general assumption for every CRL rule is that it accesses the ERT model at least once. The expression that actually accesses the ERT model is named *static class reference*. Every static class reference selects a (sub)set of instances of an entity class or a value class of the ERT model. For example, if figure 3 depicts the relationship between the entities EMPLOYEE and DEPARTMENT, then the static class reference

```
EMPLOYEE works_for DEPARTMENT
```

selects all the employees that work for a department, thus returning the set [E_1,E_2,E_4,E_5].
If, instead of accessing the employees, we wished to find the departments for which at least one employee works for, we should write

```
DEPARTMENT has EMPLOYEE
```

which would return the set [D_1, D_2, D_3]. A static class reference returns set of instances of the first entity or value class in the expression.

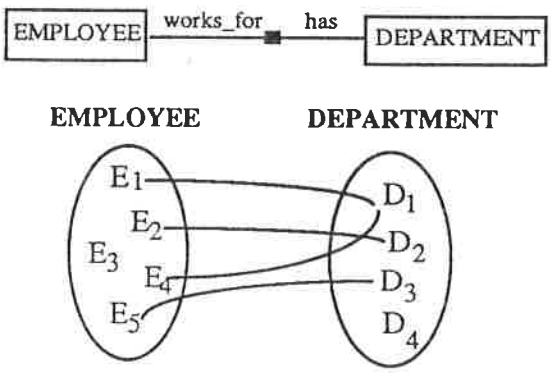

Figure 3: Example ERT schema and instantiation of it

Entity and value classes may be instantiated in a static class reference in which case they are not treated as sets but as instances of sets (i.e., every instance of the set returned by the static class reference is treated individually). Instantiation binds instances of sets when the same instance is used more than once in the same constraint rule. For example, if the relationship of figure 3 is considered, the static class reference

```
EMPLOYEE works_for DEPARTMENT.D
```

can be interpreted as "For every Department find the Employees that work for it" and the result will be the sets [E_1, E_4], [E_2], [E_5] instead of [E_1,E_2, E_4, E_5] which would be the result of the same expression without the instantiation of Department. An example of how instantiation can be used in a rule is given below:

```
EMPLOYEE works_for DEPARTMENT.D is_disjoint_from MANAGER manages
DEPARTMENT.D
```

This rule states that "for every Department, the set of Employees that work for it must be

different from the set of Managers that manage this particular Department".

Different ways to access an ERT diagram

There exist two different kinds of static class reference. The first one can be used when every entity or value classes in the static class reference is referenced only once. Example static class references of this kind are given below:

```
(i)    EMPLOYEE works_for DEPARTMENT is_managed_by MANAGER
(ii)   EMPLOYEE works_for DEPARTMENT is_managed_by MANAGER has Name where
       Name = 'Jones'
```

It should be stressed that in static class references of this kind, involvement names refer to the entity or value classes that are nearest to them, i.e.,in example (ii) is_managed_by is an involvement between Department and Manager rather than between Employee and Manager.

The second type of static class reference can be used when more than one references are made to the same entity or value class. Example static class references of this kind are given below:

```
(i)    EMPLOYEE [works_for DEPARTMENT | is_employed_in COMPANY]
(ii)   ARTICLE_GROUP [ has Account, has Name, has Number where Number =
       750]
```

It should be noted that in static class references of the kind shown above, role names inside the brackets refer to the entity or value class outside the brackets. The two connectors "|" and "," stand for union and intersection of sets and they have been included in the grammar for easiness.

4.2.2 Constraint Rules

Constraint rules express restrictions on the ERT components. Two types of constraint rules are distinguished: static constraints and transition constraints. *Static constraints* apply to every state of the database and are time-independent. They represent conditions which must hold at every state of the database. *Transition constraints* define valid state transitions in the database, thus specifying restrictions on the behaviour of the system. A constraint rule either holds, in which case it returns TRUE, or is violated, in which case it returns FALSE. Constraint rules can be grouped together forming more complex rules by using the three logic connectives *AND*, *OR*, *NOT*. For example, the rule

```
PRICE_REGULATION_ARTICLE has Net_Price is_disjoint_from
PRICE_REGULATION_ARTICLE has Reduction_Amount  AND
PRICE_REGULATION_ARTICLE has Reduction_Amount is disjoint_from
PRICE_REGULATION_ARTICLE has Reduction_Percentage
```

holds if both of its sub-rules hold. In case that one of the sub-rules returns FALSE (i.e. it is violated), then the whole constraint is violated.

Static constraints

In essence, there are three different ways for writing a static constraint rule:

1. Comparing sets of ERT data:

```
i.e.  CUSTOMER has Corporate_Code is_subset_of CUSTOMER has
Organisational_Number
```

2. Comparing enumerated sets of ERT data

i.e.

```
Number_of (OPEN_ACCOUNT bills PREMISE) <= 1

Number_of (ARRANGEMENT [governs ESB_ACCOUNT, has Status where Status =
```

```
"current"]) <= 1

Number_of (INSTALLMENT comprises ARRANGEMENT.A) < 50
```

3. Restricting value classes

i.e. `Salary is_of EMPLOYEE works_for DEPARTMENT has Name where Name = "Computation" >= 10000`

Transition constraints

The objective of transition constraint rules is to provide the means for describing business rules that refer to the past or the future. For example, consider a rule stating that a company decides (now) that it will never allow a reduction of its number of employees. Such a rule enforces the number of employees in past state (from now) to be less or equal to the number of employees in any future state of the company (and eventually the database). Such types of constraints do not refer to the current state of the database; they rather reflect a part of business policy across time and they represent the connections between database states, as they develop across time. CRL supports both implicit and explicit temporal connectives to be used as part of the transition constraints. Combinations of the implicit temporal connectives are provided, while intervals and time points along with operations (and comparisons) on them are used for defining explicit references to time (explicit temporal connectives). Some examples are given below:

```
always_in_the_future  always_in_the_past Salary is_of EMPLOYEE > 10,000  (1)

EMPLOYEE works_for DEPARTMENT at [start_of_today, X] is equal_to EMPLOYEE
at [start_of_today,X]  (4)
```

Rule (1) states that in every state of the business world the salary of an employee for every previous state was greater than £10K. Rule (4) states that starting from today until X all employees must work for department.

4.2.3 Derivation Rules

The purpose of derivation rules is to specify a way by which one can obtain the value of an ERT component when needed, instead of having it explicitly stored. For example, consider the ERT part of the figure 4:

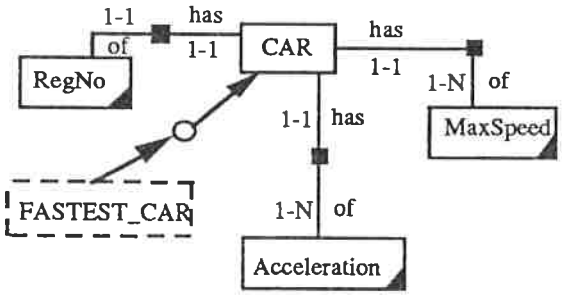

Figure 4: Example ERT Schema

According to this schema, the derived entity class FASTEST_CAR could be defined as follows:

```
FASTEST_CAR(C) is_derived_as
CAR[has MaxSpeed(S), has Acceleration(A)] and S ≥ maximum({S1|Car has
MaxSpeed(S1)}) and A ≤ minimum({A1|Car has Acceleration(A1)})
```

In the above example, the instances of the derived entity class FASTEST_CAR are defined from instances of the entity class CAR as the cars that have the maximum possible speed and the

minimum acceleration.

4.2.4 Event-action Rules

Event-action rules express reactions when certain situations occur; they control the processes that the system will be required to perform. The situations correspond to *events* either perceived in the environment or in the system itself, and to *conditions* (preconditions) that must hold on the model. This reaction corresponds to *postconditions* the achievement of which will be through processes performed onto a system or forwarded to the environment. The general format of an event-action (e-a) rule is as follows:

WHEN <event>
[IF <precondition>]
THEN <postcondition>

and described as WHEN an event is detected, IF a certain condition holds, THEN a new system state satisfying the postcondition will be the result. An event-action rule is applied on a schema's view and manipulates (not necessarily modifies) objects of that view. Data propagate in the rule, from the WHEN part down to the THEN part. However, each component of the rule is assumed to be an individual construct and no communication exists between them.

4.2.4.1 WHEN part

The WHEN part of an e-a rule corresponds to an event, which if detected, the rule is triggered. Events are of three types: *external*, *internal* and *temporal*.

External Events: External events are *happenings perceived in the environment and have some importance for the system* [Rolland and Richard 1982], [Brinkkemper 1990]. An external event has a name and a number of parameters, each one of which corresponds to some object in the model. The structure of an external event is:

$$\text{external\_event\_name}(\text{parameter}_1, ..., \text{parameter}_n)$$

where parameter_i refers to an object. Examples of external events are:

(i) Reservation_request(Departure_airport,Destination_airport, Departure_date)
(ii) Update_budget_of_department(Department_name)
(iii) Employ_new_employee(Name,Address,Tel_nr,Experience,Age)

Internal Events: Internal events are *state transitions* [Rolland and Richard 1982], . It is necessary to emphasise that the opposite does not always hold; not all state transitions are internal events. We are only interested in those state transitions that have some effect onto the system. Therefore, *a state transition is an internal event iff it causes the activation of at least one event-action rule*. A state transition is caused by any operation that causes a change in the population of a model . This defines the association between these operations and internal events:

operation_causing_change *may_cause* **internal event**

An internal event is expressed in terms of the way this event is to be consumed by the system. If, for example, the system must react whenever a certain operation is performed, then the internal event is that operation. If, on the contrary, the system must react whenever an operation of this kind and a certain situation has occurred, then both the operation and the situation comprise the internal event. More that one rules may have the same internal event. Internal events like the one in the second category , have semantics similar to pre- and post- conditions that denote a state transition.

Temporal Events: Temporal events are special types of internal events that refer to time. Examples of temporal events are:

(i) 12:00
(ii) end_of_the_month

4.2.4.2 IF part

The IF part of an e-a rule expresses a condition on the model. Once the rule is triggered, this condition must be true for the action (specified in the THEN part) to take place. The IF part of a rule may be omitted, in which case the condition is supposed to be TRUE. The condition on the model is expressed similarly to the constraint rules described in section 3.

4.2.4.3 THEN part

The THEN part of an event-action rule expresses the actions to be performed after the firing of an event-action rule. The action described in the THEN part describe how to reach a new state of the model, a state transition. This implies that at least one fact is modified after the execution of an event-action rule.

External Actions: External actions are results and/or messages forwarded to the external world. They always emit the model's data retrieved or modified by the rule. These data are considered as parameters of the external action:

<external_action_name> (parameter$_1$, ..., parameter$_n$)

An example of external action is the following, in which the parameters (Flight_no etc) are assumed to be retrieved by the rule:

Flight_and_Seat_Details (Flight_no, Depart_date, Depart_airport, Destin_airport, Depart_time, Avail_economy, Avail_business, Avail_First)

Arithmetic Operations: Arithmetic operations include any operation that manipulates data, including standard arithmetic operations (addition, subtraction etc), aggregate operations, etc. An arithmetic operation always refers to a schema's objects, thus not allowing operations of the type 1+2. The arithmetic operation is performed on an expression and the result is assigned into a variable, e.g., New_salary := instance_of(Salary is _of EMPLOYEE). The format of arithmetic operations is summarised below:

```
<variable> := <arithmetic_operation_on_ert>
```

where arithmetic_operation_on_ert is any arithmetic operation allowed in a condition.

5. Why Deductive Capabilities were Required for the Case Study

It was observed that the particular application required a certain amount of intentional knowledge to be represented in order to be able to derive new information from the facts stored in the database.

For example, some types of tracks might be considered more dangerous than others and this has to be reflected in the priority assigned to each one of them in order for the ship to defend against them firstly. In addition, certain weapons are considered better than others in defending against particular tracks. It is also known beforehand that certain defence weapons cannot be used in all situations (i.e. a gun located on the right hand side of OWN_SHIP cannot be fully moved and therefore cannot be used to defend against an attack coming from the opposite side of the ship or a torpedo is useless against an aircraft). Finally, different types of weapon systems need different times in order to be prepared for further use.

The first kind of knowledge is to be used in deciding a track's priority, the second in allocating the best of the available weapons against a track and the third one will exclude from the available weapons the ones that are not usable against a certain track's attack but may be used to defend against others. The last one is needed when deciding if a weapon that is currently in use can be considered -as part of the plan- as a later time option to defend against a track.

The types of knowledge described in the first paragraph of this section are known even before the database for the application domain is created. They are more easily and naturally represented in a deductive environment where they will be implemented either as facts (not stored in the database) or rules. Additionally they can be externalised from the actual programming code and thus become easy to handle and maintain.

Other information can be derived from the facts that are stored in the database and need not be explicitly stored. For example, the priority of a track is determined by the environment in which the track is located, the track's type, make and distance from the ship. The fact that some tracks are more dangerous than others is represented as part of the intentional knowledge but the actual priority assigned to a track will be determined by applying the predicates and rules of the intentional knowledge to the facts that are stored in the database. In a traditional database system (i.e. relational) these parts would have to be implemented in a more complicated manner as part of the actual application programming code (i.e. with the use of multiple 'case' statements to distinguish between different priorities assigned to different types of tracks) where in a deductive environment the user has only to supply the rules and the system will do the matching.

5. The Implementation Platform: MegaLog

The MegaLog persistent programming environment has been developed within the Knowledge Bases Group at the European Computer-Industry Research Centre (ECRC) and was designed as a support platform for the next generation of database management systems (DBMSs). MegaLog offers a deductive database system, that is, a system with all the functionalities of a traditional DBMS but enhanced with an additional, integral, deductive capability.

MegaLog supports in an entirely transparent and symmetric way the persistent (disk) storage and the access and update of rules as well as data [Horsfield et al, 1990]. As a result the number of stored rules is not any more limited by the available main memory. Newly acquired rules are interactively and incrementally compiled into a rellocatable form in order to avoid overheads of interpretation on execution. Transaction handling, along with recovery and concurrent operations in a multi-user environment are also provided.

The environment is based on an abstract machine containing three major components:

- an inference engine based on a derivative of the Warren Abstract Machine (WAM),
- a relational engine that has been built over a file management system and is capable of directly indexing complex objects and logical clausal structures,
- a main memory management subsystem performing full incremental garbage collection and providing full support for dynamic allocation of memory on demand.

The design of MegaLog has been based on the observation that unification in logic can be viewed as an extension of addressing by content in traditional database systems. There is no impedance mismatch problem between the database language and the programming language because there is no distinction between the two of them in a persistent logic programming environment.

The programming language interface is a full implementation of PROLOG extended with arrays and global variables and with an integrated set of relational operations. Both tuple and set oriented operations are provided. MegaLog can, therefore, support a relational system as a special case of its more generalised logic formalism. The incorporation of the garbage collector fulfils a requirement that is fundamental to database systems -continuous operation- which is however not normally supported by PROLOG systems.

The efficiency of the system is claimed to be of the same order as that of current commercial relational systems, for the same class of relational operations. Beyond that, we know of no

other that is widely available at the moment and can offer comparable functionality.

6. The Transformation from the Conceptual Level to the Deductive Platform

Mapping ERT to MegaLog

The algorithm for transforming an ERT schema to the deductive component is described in the following steps:

Step 1 Every simple entity is represented as a predicate having as arguments all the entity's involvements with value classes which in turn will include these classes that appear on the ERT model. If the entity has complex attributes go to step 2, if it is timestamped go to step 3 and if there exists an ISA relationship for it on the ERT go to step 4.

Step 2 Complex attributes have the same form (predicates with arguments all the simple attributes) as entities, but they are not represented explicitly; they appear only as part of the entity to which they belong.

Step 3 When a timestamp appears as part of an entity or relationship add a new part to the entity or relationship that will have the form time(start_time, end_time).

Step 4 Depending if the ISA relationship represents a partial disjoint, a total disjoint, a partial overlap or a total overlap perform steps 4a, 4b, 4c or 4d respectively:

- **Step 4.a** Add a predicate for the entity as in step 1 and also add predicates for it describing what it can be and what it is not allowed to be at the same time.
- **Step 4.b** Add only the predicates for the entity showing what it can be and it is not allowed to be at the same time.
- **Step 4.c** Add a predicate for the entity as in step 1 and also add predicates for it describing what it can be.
- **Step 4.d** Add only the predicates for the entity showing what it can be.

Step 5 The assertion that two entities are connected through a relationship is represented by a predicate having as arguments the keys of these entities. The validated fact assertion depends on the existence of the connected entities.

Step 6 Complex entities are represented as simple entities where every simple entity that appears within the complex one has the same form as a complex attribute; the only difference is that because of rules 1 and 5 and simple entities and the relationships (apart from relationships indicating attributes) between them are represented explicitly too.

Step 7 When a uniqueness or cardinality constraint appears on the ERT add a new clause in the entity's or relationship's assertion that will not permit more than the allowed number of same values to exist. These constraints are to be checked every time upon insertion, deletion and update.

The results of the mapping for the entities Own_ship and Track presented in Figure 5 is as follows (without representing the cardinality constraints):

own_ship(has(Longitude, time(Start_time, End_time)), has(Latitude, time(Start_time, End_time)), has(Course,time(Start_time,End_time)), has(Speed,time(Start_time,End_time)), has(ship_id(hasComponent(...),hasComponent(...)...),has(power(hasComponent(...),hasComponent(...)...), has(propulsion(hasComponent(...),hasComponent(...)...)).

track(has(Iff_code), has(Id), has (Detection_time), is_located_in(Environment), has(Longitude, time(Start_time, End_time)),has(Latitude, time(Start_time, End_time)), has(Speed,time(Start_time,End_time)), has(quality_of_info(hasComponent(...),hasComponent(...)...), has(track_type(hasComponent(...),hasComponent(...)...)).

Mapping from CRL to MegaLog

Every static constraint is composed by multiple accesses to an ERT diagram which are connected via set operators (union, intersection, difference) or set comparisons (subset_of, disjoint_from, equal_to). The first way in which an ERT access can be performed is when every entity or value is referenced only once, i.e.:

CUSTOMER responsible_for CLAIM is_based_on PRICE_REGULATING_AGREEMENT contains PRICE_REGULATION_GROUPRATING refers_to GROUPRATED_POSTAL_ITEM

which will return a set of all customers working for which all of the above apply.
Such an ERT access consists of triplets that in the simplest of cases have the form

ENTITY relationship ENTITY

or VALUE in the place of entities, but no two values can exist in the same triplet. Each triplet can be represented with three clauses -one for every part of the triplet- within a predicate that will return the required result. The last part of a triplet can be the first one of the next, as is the case in the previous example, in which case the clause representing it need not be repeated. The ERT access will return a set containing all existing entities (or values) of the type that is referenced first within the ERT access; in the example the result will be a set of employees. Therefore the key of the first entity (or the first value) will be the argument of the predicate that will be used, i.e.

```
ert_access(Customer_id):-
        customer(Customer_id,_,_,...),
        responsible_for(Customer_id, Claim_id),
        claim(Claim_id,_,_,...),
        is_based_on(Claim_id,Pr_reg_agr_id),
        Pr_reg_agr(Pr_reg_agr_id, _,_,...),
        contains(Pr_reg_agr_id, pr_reg_group_id),
        pr_reg_group(pr_reg_group_id,_,_,...),
        refers_to(pr_reg_group_id,gr_rat_p_item_id),
        gr_rat_p_item(gr_rat_p_item_id,_,_,...).
```

The above predicate will return every customer that satisfies the selected criteria; because a set is expected another predicate will be needed to collect the results of the above one:

```
ert_access1(Result):-
findall(Customer_id, ert_access(Customer_id), Result).
```

If the ERT access requested a value instead of entity the value would be requested both as the output of the first predicate (also it has to be referred as a variable in the owner's entity clause) and in the collecting statement.
When a value appears in an ERT access the value and the relationship ('has' or its inverse 'is_of') connecting it to the 'owner' entity are ignored and the three clauses are reduced to just one, that of the 'owner' entity. If the value class is followed by a 'where ...' statement with an equality expression, then the value is inserted as a constant in the entity's predicate. For example, the simple ERT access

ARTICLE_GROUP has Number where Number = 750

will be represented as

article_group(Article_group_id,_,..., 750,...).

In any other case the value class will be inserted as a variable in the entity's clause and if the 'where ...' statement includes any other comparison except equality a new clause will be added to restrict the value, i.e.:

ARTICLE_GROUP has Number ==> article_group(Article_group_id,_,..., Number,...)

ARTICLE_GROUP has Number where Number >= 750
==> article_group(Article_group_id,_,..., Number,...)
Number >= 750

In the case where an entity or value class is referenced more than once in an ERT access the treatment is the same as before bearing in mind that role names refer to what is outside the brackets and therefore the first part of several triplets will be the same. If, however, an OR is reached within the brackets then more predicates will be required: one for representing everything until the bracket in which the '|' is found and as many others as the unioned expressions are. The first predicate will call all the others which will have the same name and same argument that will be the part just outside the bracket (because the part outside the brackets is the same for all expressions included in them). For example, the ERT access

AGREEMENT is_valid for ARTICLE_GROUP [has Account, has Name, has Number where Number = 750 | has Type where Type ='aaa']

will be represented as

```
new_acess(Agreement_id):-
agreement(Agreement_id,_,...,_),
is_valid_for(Agreement_id, Article_group_id),
new_access1(Article_group_id).

new_access1(Article_group_id):-
article_group(Article_group_id,...,Account,...,Name,...,..., 750,...).

new_access1(Article_group_id):-
article_group(Article_group_id,...,aaa,...).
```

In cases of instantiations the clauses that should have been included first and second(the one representing the target entity or value and the relationship) will be substituted by a collecting statement for the first clause on the first relationship appearing in the ERT access and will be inserted at the end of the predicate. This happens because of interest are the results for every instantiation and not for every entity or value class that participates in the ERT access. For example,

INSTALLMENT comprises AGREEMENT.A

is requested to return for every agreement the set of installments comprising it and therefore it will be represented as:

```
third_ert_access(Result):-
agreement(Agreement_id, _,...),
findall(Installment_id,comprises(Installment_id,
Agreement_id), Result).
```

If the constraint refers to enumerated sets of ERT data then the result of the collection of values an aggregate operator (number_of, average etc.) that is easily implemented will be applied. For example, in the example that was used in the beginning of the paragraph if the ERT access was of the form

number_of(INSTALLMENT comprises AGREEMENT)

the ert_access would have the form:

```
last_ert_access1(Final_Result):-
findall(Installment_id, ert_access(Installment_id), Result),
number_of (Result, Final_Result).
```

Implemented explicit or implicit temporal operators are applied to the results of the ERT accesses and then two ERT accesses are compared or transformed via set operators or set comparisons. The representation of these is not difficult and the result will be

ERT access1
set comparison or set operation
ERT access2

which will result in a static constraint having the form:

constraint(Result):-
ert_access1(Result1),
ert_access2(Result2),
perform_operation(Result1, Result2, Result).

Derivation rules use an ERT access format to define a object and therefore their representation in MegaLog is straightforward. Every time that is needed a predicate that will return the derived object will be called to perform the specific ERT access that will calculate the derived object's value.

For the event-action rules we require one predicate for each rule which in turn will be comprised by three others. The first one of these three will either be a system call (to determine whether or not an external event happened) or a checking predicate upon an affected relation (for internal events). The second one will be the precondition part of the event-action rule which is treated in the same way as static constraints and the third one defines the action to be taken as a consequence of the firing of the rule. This will eventually correspond to a process that will need to be implemented.

The above considerations lead to the following algorithm for mapping CRL to MegaLog:

Step 1: If derivation rule go to step 5
else
IF event-action rule go to step 6
else
Step 1.1: Split constraint in ERT accesses (these are the parts before and after the set operations)..If the ERT access is a simple one go to step 2 else go to step 3.

Step 2: Simple ERT access: split ERT access in triplets and
Step 2.1: Include clauses for each triplet not repeating clauses that already exist.
Step 2.2: If value class appears ignore the value class and its relationship to the entity to which it belongs and if:
Step 2.2.1: If value class is followed by a 'where' statement with equality expression insert constant in the value's place within the 'owner' entity's clause,
else
Step 2.2.2: If a 'where' statement follows without equality then use a variable in the value's place and add a clause restricting its value according to the 'where' part of the ERT access,
else
Step 2.2.3: Simply add variable in the value's place in the entity's clause.
Step 2.3: If value is requested it has to be the predicate's argument and referred in the owner's entity clause as a variable.
Step 2.4: Go to step 4.

Step 3: Complex ERT access: split in triplets and
Step 3.1: Include clauses for each triplet avoiding repetition of clauses and in case of brackets the first part of the triplet is the one just outside the brackets
Step 3.2: If all triplets are connected via AND (',') then continue as before
else
If an OR ('|') is reached split ERT access predicate in two or more from which the first one will containing all entities, relationships and value classes until and not included the one outside the brackets. For the remaining part of the ERT access create as many predicates as the unioned parts having the same name and argument (the part just outside the brackets); these predicates will be called by the first one.

Step 3.3: If value class appears ignore the value class and its relationship to the entity to which it belongs and if (same as for step 2):

Step 3.3.1: If value class is followed by a 'where' statement with equality expression insert constant in the value's place within the 'owner' entity's clause,
else

Step 3.3.2: If a 'where' statement follows without equality then use a variable in the value's place and add a clause restricting its value according to the 'where' part of the ERT access, else

Step 3.3.3: Simply add variable in the value's place in the entity's clause.

Step 3.4: If value is requested it has to be the predicate's argument and referred in the owner's entity clause as a variable(same as 2.3).

Step 3.5: Go to step 4.

Step 4:

Step 4.1: If no instantiation is found in the ERT accesses then collect for each ERT access in the constraint all values and

Step 4.1.1: If constraint applies to enumerated sets of ERT data apply aggregate operators to resulting sets and if the constraint is a transition one apply the temporal operators as well
perform operation or comparison to the resulting sets
else

Step 4.2: Remove first two clauses of ERT access, apply temporal operators, if required, and add at the end collecting statement for the desired result upon the first relationship of the remaining part of the ERT access.

Step 5: Create predicate that will return the derived object and perform ERT access as in steps 2-4 to calculate the derived object's value.

Step 6: Create predicate that will have 3 other predicates as components:

a) IF external event predicate corresponds to system call
else
checking predicate upon a relation

b) For the second predicate go to Step 1.1

c) Third predicate corresponds to process.

7. Design and Implementation Issues

In this section a brief overview of the designed an implemented system is provided in terms of the main processes that the system performs, a user interface that was created for the interactive introduction of different scenarios and the statistical results of the application of the system in a real life case study.

7.1 Overview of the Prototype System

The main processes that the system is required to perform are shown in Figure 5. The decomposition of process P2 is presented in Figure 6. In both figures the rounded boxes represent the processes, the lines with the arrows the triggers (or output) of the processes and finally for each process it is shown whether or not it requires to read something from the data model.

The assumptions that were made about the knowledge that exists before the creation of any database are reflected in the intentional knowledge that was used. Four different 'sets' of intentional knowledge were represented:

(a) *Track priority:* the priority assigned to each track depends on its environment (i.e. air, sea etc.), its type (i.e. aircraft, helicopter etc.) its make (i.e. f16) and on its distance from OWN_SHIP (i.e. distance < 5000 or 5000 < distance < 10000 etc.). Each of the predicates used will -when applied to a particular track- assign a number to it. The sum of all four of them for each track will represent its overall priority.

(b) *Probability of success of a weapon against a certain type of track:* it is in the form of simple predicates that will later be used to find the best defence against the track.

(c) *Time needed for every weapon to be prepared for use:* this is used to calculate when a weapon that has currently status='occupied' will again be ready for use. It can also used for updating the track information in the meanwhile in order to define where the track will be when that particular weapon will be available again.

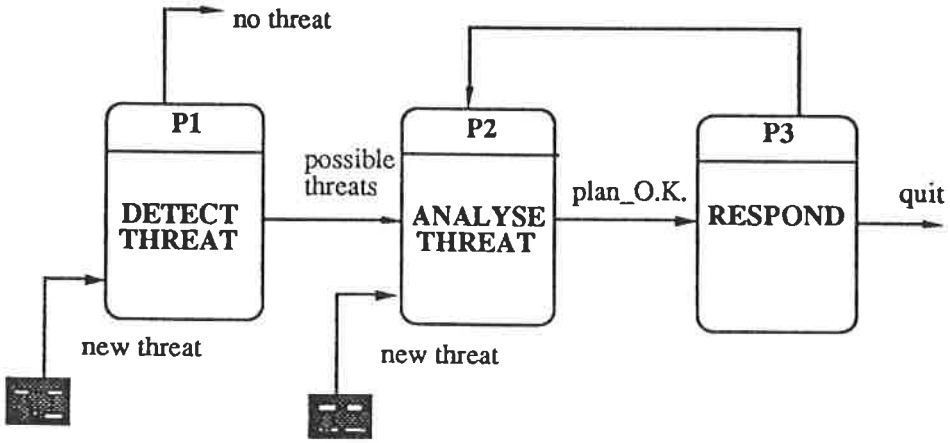

Figure 5: The main processes of the system

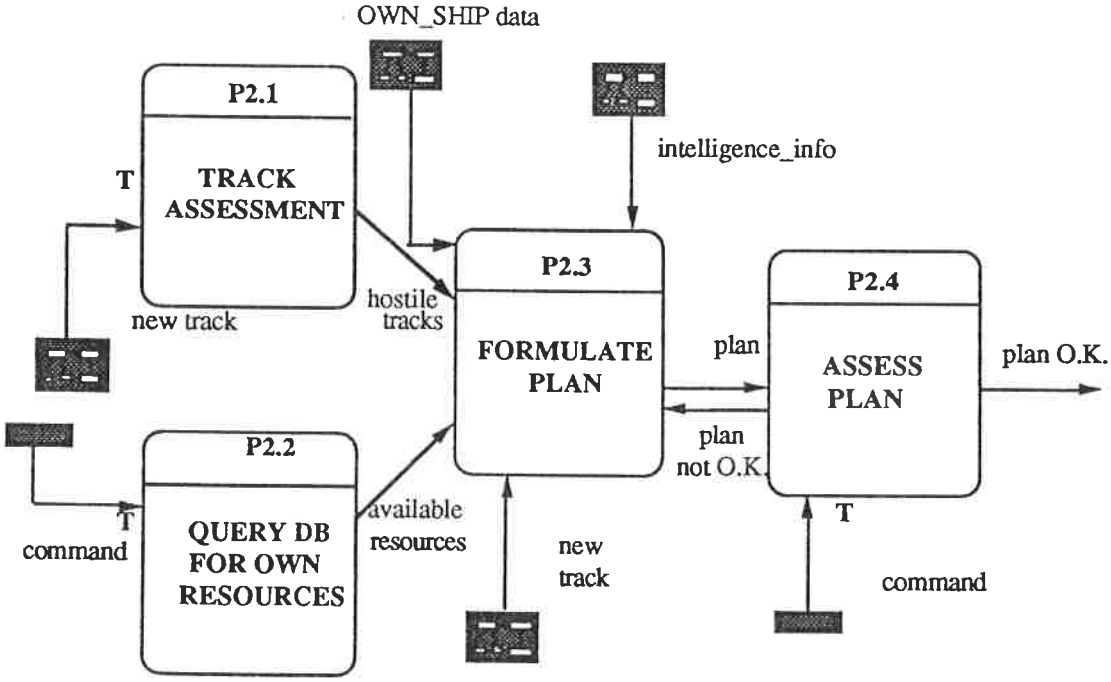

Figure 6: The decomposition of process P2

(d) *Non movable weapons:* shows the weapons that cannot be fully moved and thus their use will only be restricted to defend against attacks coming from the side of OWN_SHIP on which they are located.

The application program implementing process P2 in Figure 6 consists of three main parts each one implementing each of the three subprocesses, the first two of which can be further decomposed (the description is following a pseudolanguage):

track assessment
find hostile tracks
eliminate duplicate ones

assign priority to remaining ones
find available resources
find damaged systems
update weapon systems that were found in the list of damaged ones
create plan

The above also demonstrates and dictates in a way and the 'flow' of the application program. The action steps that comprise each one of the above subprocesses are presented below using a pseudolanguage:

find hostile track
every track for which
no iff_code is known
AND its description is 'military'
AND (OWN_SHIP latitude - X) < track latitude < (OWN_SHIP latitude + X)
AND (OWN_SHIP longitude - X) < track longitude < (OWN_SHIP longitude + X)
is considered as a hostile one.

eliminate duplicate tracks
(from hostile tracks)
get track with minimum id (i.e. X)
find all tracks that are resemble initially to X
(have the same description, environment and type with X)
find X's last two measures
find last two measures for the first track initially resembling to X
compare values:
longitude_last1 - longitude_last2 < A
AND latitude_last1 - latitude_last2 < A
AND longitude_previous1 - longitude_previous2 < A
AND latitude_previous1 - latitude_previous2 < A
AND ((longitude_last1 - longitude_previous1) / (latitude_last1 - latitude_previous1))
- ((longitude_last2 - longitude_previous2) / (latitude_last2 - latitude_previous2)) < A
(this last clause will give some kind of indication about the track's course between measures)
remove last measures for X and the first resembling track
compare rest measures for X and the first resembling track in the same way
if all comparisons succeed remove all measures for first resembling track
perform the same procedure for all the other initially resembling tracks
perform the same for all remaining tracks.

assign priority to remaining tracks
for every track get its environment
AND type
AND make
AND distance from OWN_SHIP,
get priority according to environment (i.e. X)
get priority according to type (i.e. Y)
get priority according to make (i.e. Z)
get priority according to distance from OWN_SHIP (i.e. W)
calculate overall priority as Priority = X + Y + Z + W.

find available resources
get all system damages that refer to OWN_SHIP
find which of these refer to weapon systems
for every one of these weapon systems set weapon_status = 'damaged'.

create plan
get track with highest priority
find an initial list of all usable (amongst the non damaged ones) weapons against it
find from list the weapon offering the greatest possibility of success
if weapon status ≠ 'damaged'
AND weapon status ≠ 'occupied'
AND weapon stock > 0
if weapon is characterised as 'nonmovable' check where the track comes from
if attack comes from other side discard weapon and search list for next best possibility
ELSE select weapon

```
        update weapon:          set status = 'occupied'
                                set stock = stock - 1
if no weapons can be used print 'cannot defend now against Track_id Track_type'
    find best one of  weapons with status = 'occupied' and stock > 0
    find time required to prepare weapon and update track information accordingly
    check in the same way as above if weapon is applicable in this case and
    if new track distance > DIST
        (if new track distance < DIST the track is considered to be too close to
    effectively defend against)
        THEN use weapon later
    else print 'cannot defend now against Track_id Track_type'.
```

The output of the application program will be a list containing all solutions that contribute to the overall defence plan. The resulting plan will have the form:

17 f16 missile (where the number shown is the id of the track and it is followed by its type and the weapon that is best suited for a defence against it)

Cannot defend now against 25 fr1.
25 fr1 gun (whenever these two lines appear it means that there is no defence solution applicable at that particular moment but -as can be seen in the second line- there is a later one. Whenever only the first of these two lines appears for a particular track, it means that there is no available weapon to defend against that track, not even at a later time).

7.2 A User Interface for the Creation of Different Scenarios

A user interface that is based on PCE (provided with MegaLog) has been specified and implemented for the interactive introduction of different scenarios in order to test the system's responses in a more user friendly way that does not require any PROLOG knowledge. The considerations for such a task included the ability to: display all the information concerning the ship's power, weapon systems, position, etc., as well as information about the position, type, speed, etc., of the detected tracks; create a radar type screen that will display the Playing Area of the ship and the detected tracks; also, to give the user the ability to create his/her own attack scenarios each time by changing whichever of the above described parameters he/she requires. This is done by including facilities of inserting and deleting information about the ship's position, speed, power, damages, etc. as well as for the detected tracks. This information provided interactively changes automatically the data that was already stored (or inserts new data) in the database to provide for a different scenario.

Two example layout screens, part of the interface, displaying the information about the ship's properties and all information about the ship's positioning and located tracks respectively are shown in Figure 7 and Figure 8.

7.3 Statistics, Comparison with other DBMSs, Evaluation of the Approach

The safety critical system that was described before and was implemented in MegaLog was used in a real life case study using 250 initial tracks, that were finally reduced to 25. The statistical results for this particular case study are presented in Table 1. The first case that is presented in this table corresponds to 60 initial tracks reduced to 9 while the second one to the 250 reduced to 25. In this table the time is expressed in miliseconds.

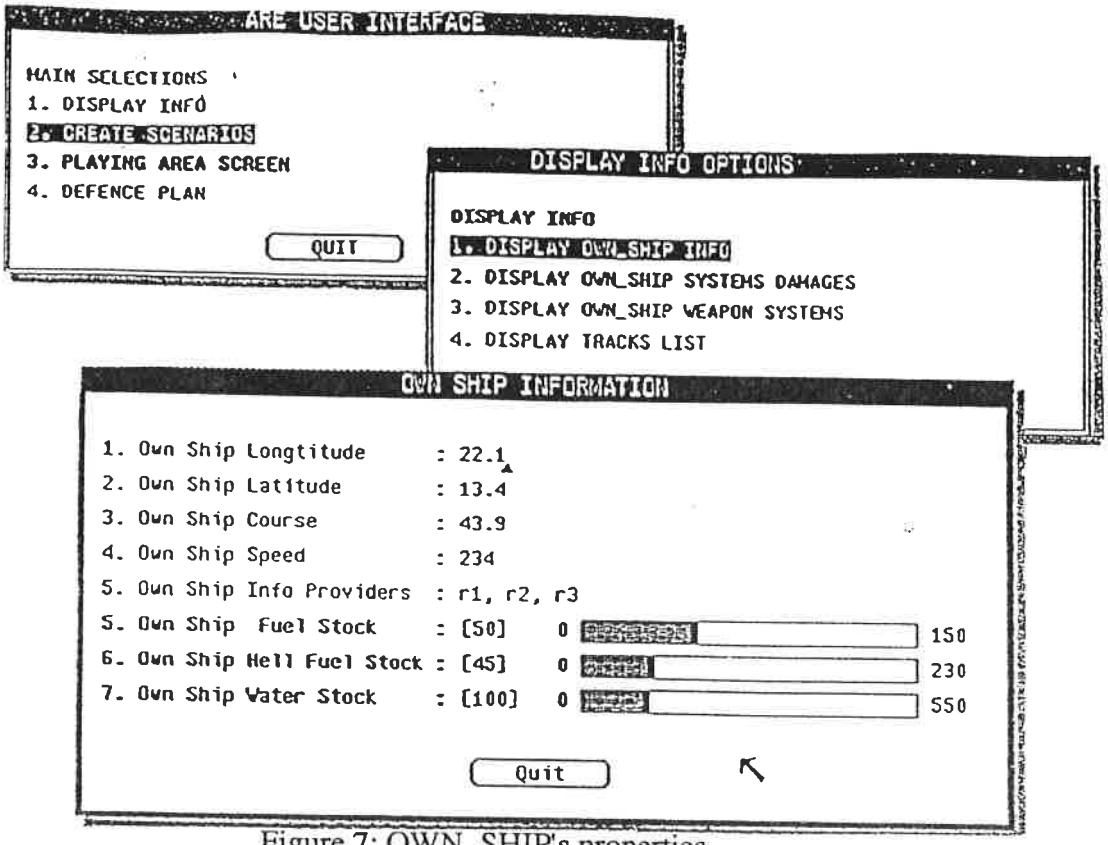

Figure 7: OWN_SHIP's properties

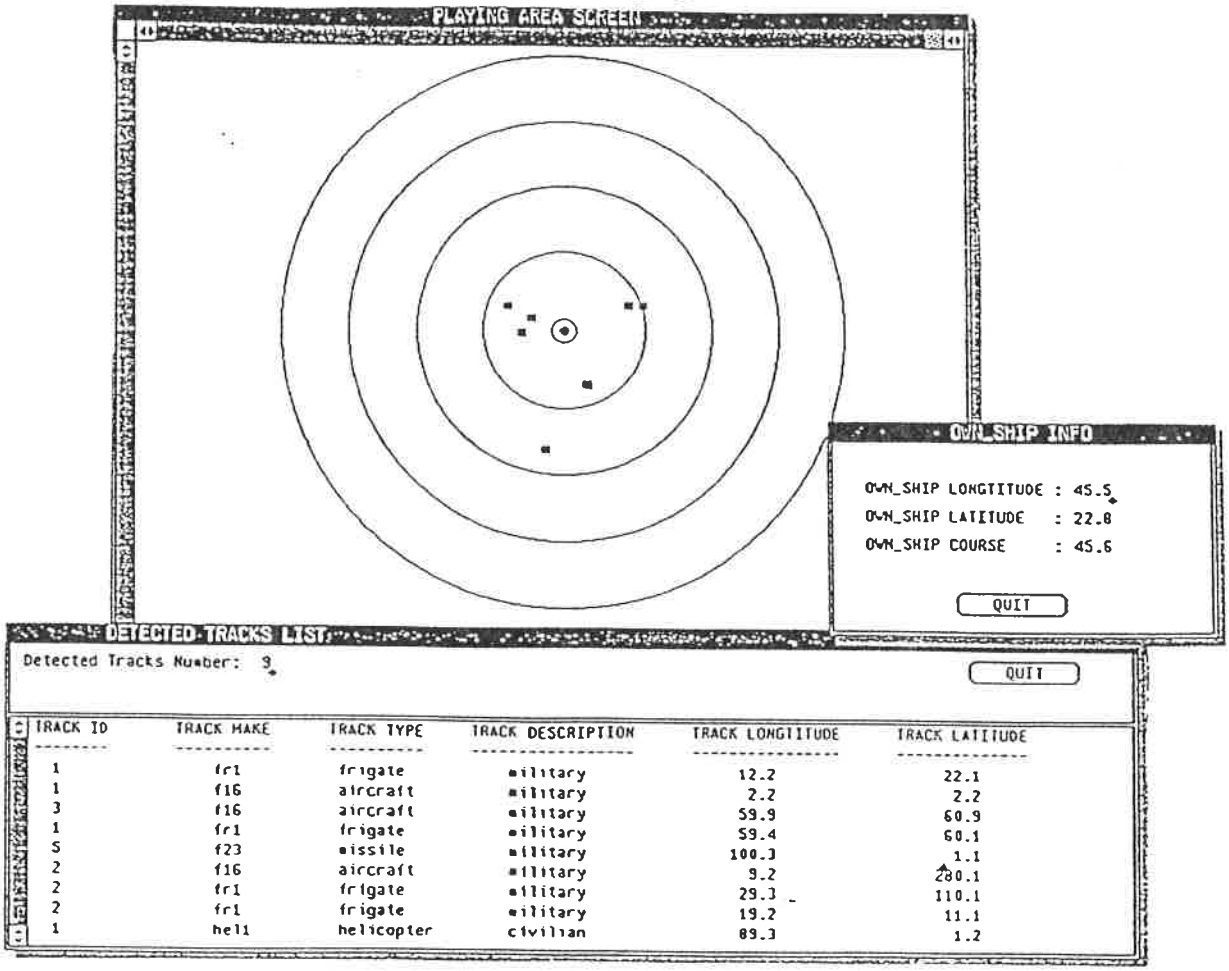

Figure 8: The ship's positioning and the located tracks

	60 tracks	250 tracks
Time Taken	74148	101400
User CPU Time	18310	33060
System CPU Time	7250	10820
Page Reclaims	1542	808
Page Faults	234	69
Swaps	0	0
Heap Statistics		
Free Memory of Buddy Algorithm		
size 2^4-4=	12 : 1 free block(s)	12 : 1 free block(s)
size 2^5-4=	28 : 51 free block(s)	28 : 4 free block(s)
size 2^6-4=	60 : 50 free block(s)	60 : 196 free block(s)
size 2^7-4=	124 : 31 free block(s)	124 : 153 free block(s)
size 2^8-4=	252 : 27 free block(s)	252 : 0 free block(s)
size 2^9-4=	508 : 17 free block(s)	508 : 1 free block(s)
size 2^10-4=	1020 : 21 free block(s)	1020 : 1 free block(s)
size 2^11-4=	2044 : 14 free block(s)	2044 : 0 free block(s)
size 2^12-4=	4092 : 2 free block(s)	4092 : 1 free block(s)
size 2^13-4=	8188 : 2 free block(s)	8188 : 1 free block(s)
size 2^14-4=	16380 : 0 free block(s)	16380 : 0 free block(s)
size 2^15-4=	32764 : 1 free block(s)	32764 : 1 free block(s)
size 2^16-4=	65532 : 1 free block(s)	65532 : 0 free block(s)
size 2^17-4=	131068 : 0 free block(s)	131068 : 1 free block(s)
size 2^18-4=	262140 : 1 free block(s)	262140 : 1 free block(s)
size 2^19-4=	524284 : 0 free block(s)	524284 : 0 free block(s)
size 2^20-4=	1048572 : 0 free block(s)	1048572 : 0 free block(s)
Memory Usage Statisitcs		
Used Memory	381801 bytes 36%	372166 bytes 35%
Waste in Used Blocks	180175 bytes 17%	178334 bytes 17%
Used for Management	27844 bytes 2%	27440 bytes 2%
Total Free Size	458756 bytes 43%	470636 bytes 44%
Blocks(free)	219	361
Blocks (used & free)	6961	6860
Segments	1	1
Symbol Table Statistics		
Valid Items in Symbol Table	2409	2293
Items added since last GC	2409	2293
Items removed by last GC	0	0
Items removed by all GC	0	0
Number of GC done so far	0	0
Hash Key Distribution		
(0,1,2,3,>3,>10)	8261 1591 311 46 14 0	8351 1523 291 44 14 0
Garbage Collection Statistics		
Number of collections of Global Stack	89	82
Total Number of Bytes Collected	8867456	8137088
Bytes Collected on Last Run	100904	100360

Table 1: Sample Statistics of the Case Study

TIME TAKEN indicates the actual time that the system required in each case in order to produce

the required defence plan. USER CPU TIME is the amount of time that the cpu spent in user mode, while SYSTEM CPU TIME is the amount of time that the cpu spent in system mode.PAGE RECLAIMS, PAGE FAULTS and SWAPS provide information about paging and swapping from the operating system (UNIX). Heap statistics include information about the usage of the heap in MegaLog while the rest indications provide information about symbol table usage and performed system table garbage collections (note that in this last case the results i.e. total number of bytes collected are added together after each run of the application, thus the number appearing in the table does not demonstrate the real situation after each time the system runs. This is achieved with the *bytes collected on last run* data).

The relationship between some aspects of the two different measurements is presented graphically in Table 2:

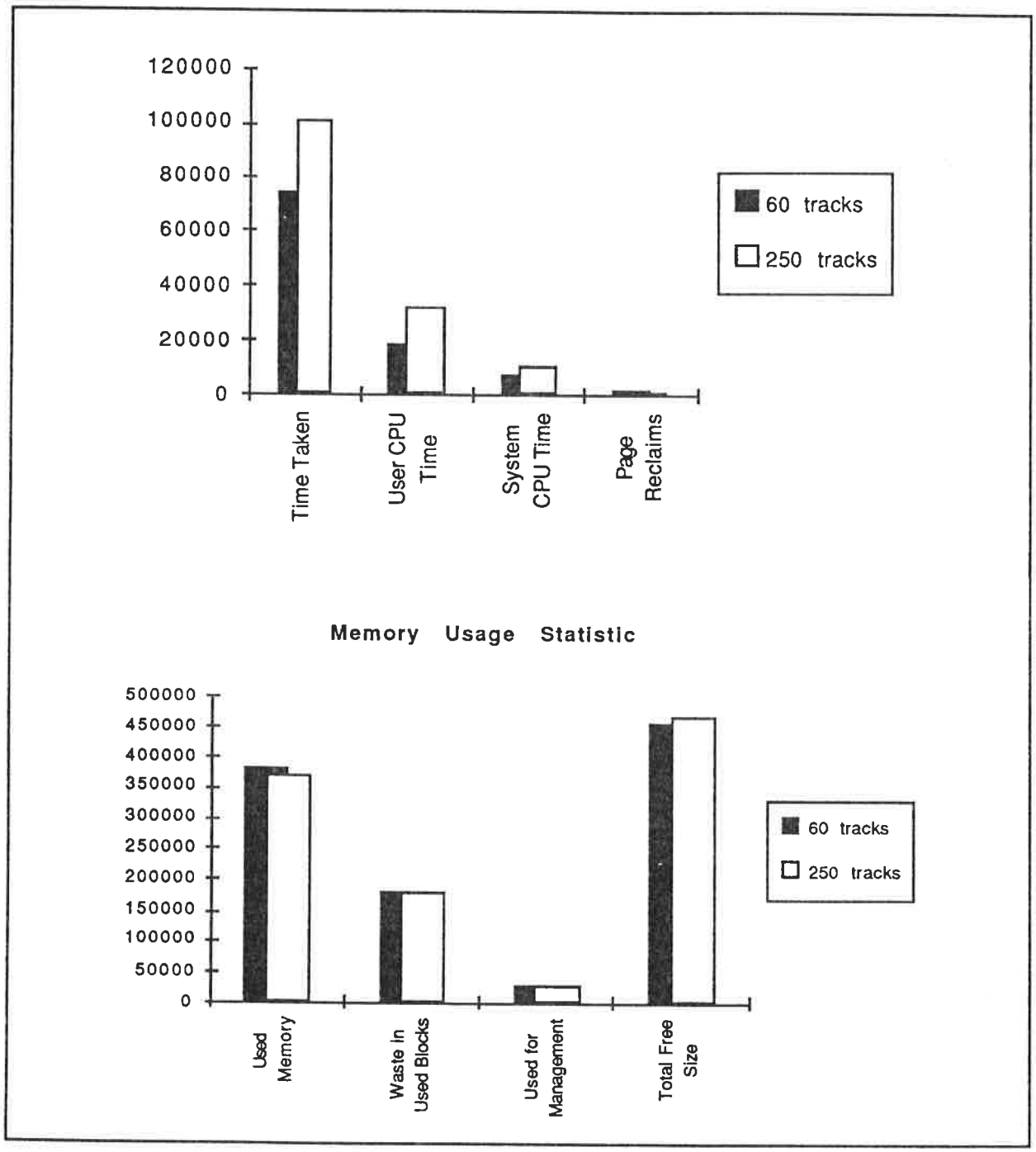

Table 2: Comparison of Measurements

These figures are comparable (in fact somewhat better) than when the same application was tried on a relational DBMS. However, the time required for the development of the system was in the case of the relational DBMS greater and especially in the parts that were identified before as those that originally required the deductive capabilities of the underlying DBMS.

Deductive database systems represent a new direction in database technology and have first order logic as their theoretical foundation. This approach has several desirable properties. Logic itself has a well-understood semantics and can be employed to provide a uniform language for defining data, programs, queries, views and integrity constraints and an operational uniformity by providing a basis for a unified attack on query optimisation, integrity constraint enforcement and program correctness proofs. It can, additionally, provide a basis for dealing with certain aspects that the relational approach has always had difficulties dealing with, i.e. disjunctive information. Deductive relations greatly extend the range of knowledge that is included in the database or derived from the base relations and reduce the size of application programs that have to be written.

In the conceptual models that were used for the safety critical system the representation of rules at the conceptual level is closer to the end user's perception. Additionally, this means that rules and constraints can be explicitly captured and maintained throughout an information system's lifecycle. They can be represented straightforward in a deductive environment and can be externalised from programming code (and kept separately in one place) thus facilitating easy access and maintenance. This is not the case in a relational or object-oriented environment where some (the simplest constraints) can be implemented as triggers or as constraints on slots and methods, but the more complicated ones will have to be part of the application's programming code, thus resulting in code redundancies and difficult to maintain application programs.

Most of the disadvantages faced during the application of the case study come from the fact that deductive technology is still more or less in a research stage. Not many commercially available products exist so that they can be tested in real life applications to fully explore their capabilities and shortcomings.

MegaLog, for example, because it is based on logic does not offer the *update* operation (which was used, for example, in updating the weapons' status and the distance of tracks from OWN_SHIP during the plan formulation) found in relational databases. This had to be implemented (as well as other operations) and this means that the application program will inevitably become slower as more operations and utilities were needed that did not form part of the actual environment.

Also the insert operation when not all values for all the attributes are supplied (in case they are not known at the time of insertion) is not supplied. In this case the handling of values becomes difficult because a default value -'null' for example- will be treated as a constant by MegaLog (and not as a special kind of value as is the case in relational systems). At the same time the insertion of variables in the place of unknown values causes greater manipulation problems, because PROLOG will match the variables with anything. This means that this 'incomplete' insert will be followed by an update (which essentially is a delete and an insert operation) and not by another insertion for the previously unknown values.

In other cases, however, the manipulation of the database contents is easier, i.e. with the provision of the *project* operation that it is not part of standard SQL.

While every relation in a MegaLog database requires at least one key to be defined, these are only used for indexing and the system will accept the insertion of a different tuple with the same value for the key. This might be an advantage in applications that do not require the usual management of primary keys (i.e. temporal databases) offered by relational systems but leads to finding artificial ways in preserving the uniqueness of key values in a relation.

8. Conclusions

The design, implementation and evaluation of an application that was carried out in a deductive environment was presented in the previous sections. The advantages and shortcomings of the approach that was followed were reported along with a comparison with relational technology.

Deductive database systems offer more expressive power than the traditional ones along with the ability to deduce new facts from explicitly stored ones. They can offer many advantages to potential users and the uniformity they provide is a very attractive idea. The field is, however, relatively new and although much research has been carried out in the area, deductive systems are still not very popular and the main reasons for that is that they have not been fully tested and that the PROLOG language they are using is not easy for first time users to understand in order to manipulate the database relations.

The MegaLog logic programming environment with some additional relational operations (i.e. updates, inserts with not all values provided) was proven very useful in the application of the case study that was chosen and it is one of the few products that are widely available - despite some shortcomings - to offer both database and logic programming functionalities integrated.

The application of the conceptual models that were used in a deductive environment demonstrates that, because the processes, their control and the checking for the consistency of data were be represented as rules and goals and kept outside programming code, application programs can become smaller and easier to write and to maintain. The interest currently is in extending the deductive platform with distributed capabilities. It is anticipated that minimising the duplication of processes and especially the huge amount of data for the particular application that was described will greatly enhance the system's performance.

References

[Ahn & Snodgrass, 1988] Ahn I., Snodgrass R., *Partitioned Storage for Temporal Databases Information Systems*, 13(4), 1988.

[Ariav & Clifford, 1984] Ariav G., Clifford J., *A System Architecture for Temporally Oriented Data Management*, Proceedings of the 5th International Conference on Information Systems, Tucson Arizona, Nov.1984.

[Batini, 1988] Batini, C. and Di Battiste G. *A Methodology for Conceptual Documentation and Maintenance*, Information Systems, 13(3), pp.297-318, April 1988.

[Ben-Zvi, 1982] Ben-Zvi J., The Time Relational Model , PhD Dissertation, Univ. of California, L.A., 1982.

[Chen, 1976] Chen P.P-C. *The Entity-Relationship Model-Toward a Unified View of Data* ACM TODS vol.1 no.1, pp.9-36, March 1976.

[Brinkkemper, 1990] Brinkkemper, S.B.,*Formalisation of Information Systems Modelling*, Univ. of Nijmegen, The Netherlands.

[Dadam et al, 1984] Dadam P., Lum V., Werner H.D., *Integration of time versions into a relational database system*, Proc. VLDB, Singapore, 1984.

[Date, 1990] Date, C.J. *An Introduction to Database Systems*, Volume I, 5th Edition, Addison-Wesley Publishing Company, Inc., 1990.

[Elmasri and Navathe, 1989] Elmasri, R. and Navathe, S.B. *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company, Inc. 1989.

[Gallaire and Minker, 1978] Gallaire, H. and Minker, J. (eds) *Logic and Databases*, Plenum Press, New York 1978.

[Gallaire et al, 1980] Gallaire, H., Minker, J. and Nicolas, J.M. *Background for Advances in*

Database Theory, in Advances in Database Theory Volume 1, Gallaire, H., Minker, J. and Nicolas, J.M. (eds), Plenum Press, New York, 1980.

[Gallaire et al, 1984] Gallaire, H., Minker, J. and Nicolas, J.M. *Logic and Databases: A Deductive Approach*, Computing Surveys, Vol 16., No. 2, June 1984.

[Gardarin and Valduriez, 1989] Gardarin, G. and Valduriez, P. *Relational Databases and Knowledge Bases*, Addison-Wesley Publishing Company, Inc., 1989

[Horsfield et al, 1990] Horsfield, T., Bocca, J. and Dahmen, M.*MegaLog User Guide*, October 1990.

[Kim et al, 1987] Kim W., Banerjee J., Chou H.T., Garza J.F., Woelk D. *Composite Object Support in Object-Oriented Database Systems*, in Proc. 2nd Int. Conf. on Object-Oriented Programming Systems, Languages and Applications, Orlando, Florida, Oct. 1987.

[Kent, 1979] Kent W. *Limitations of Record-Based Information Models*, TODS, 1979.

[Lorie et al, 1983] Lorie R., Plouffe W. *Complex Objects and Their Use in Design Transactions*, in Proc. Databases for Engineering Applications, Database Week 1983 (ACM), San Jose, Calif., May 1983.

[Lloyd and Topor, 1985] Lloyd, J.W. and Topor, B.W. *A Basis for Deductive Database Systems*, The Journal of Logic Programming, 1985(2), pp 93-109.

[Lloyd and Topor, 1986] Lloyd, J.W. and Topor, B.W. *A Basis for Deductive Database Systems II*, The Journal of Logic Programming, 1986(5), pp 55-67.

[Lum et al, 1984] Lum V., Dadam P., Erbe R., Guenauer J., Pistor P., *Design of an integrated DBMS to support advanced applications*, Proc. Conf. Foundation Data Organization, Kyoto, Japan, 1985.

[McKenzie, 1986] McKenzie E., *Bibliography : Temporal Databases*, ACM SIGMOD, Vol.15, No.4, December 1986.

[Minker, 1988] Minker, J. (ed). Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann Publishers, Inc., 1988.

[Rabitti et al, 1988] Rabitti F., Woelk D., Kin W. *A Model of Authorization for Object-Oriented and Semantic Databases*, in Proc. Int. Conf. on Extending Database Technology, Venice, Italy, March 1988.

[Nijssen et al, 1988] Nijssen G.M., Duke D.J., Twine S.M.*The Entity-Relationship Data Model Considered Harmful*, 6th Symposium on Empirical Foundations of Information and Software Sciences, Atlanta, Georgia (USA), October 1988.

[Reiter, 1984] Reiter, R. *Towards a Logical Reconstruction of Relational Database Theory*, in On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages, Brodie, M.L., Mylopoulos, J. and Schmidt, J.W. (eds), Springer-Verlag, New York, 1984.

[Rolland and Richard, 1982] Rolland, C. and C. Richard, *The REMORA Methodology for Information Systems Development and Management*, Conference on Comparative Review of Information System Design Methodologies, North Holland,

[Theodoulidis & Loucopoulos, 1991] Theodoulidis, C. and Loucopoulos, P. *The Time Dimesion in Conceptual Modelling*, Information Systems, 16(3), 1991.

[Theodoulidis et al, 1990] Theodoulidis, C., Wangler, B. and Loucopoulos, P. *Requirements Specification in TEMPORA*, 2nd Nordic Conference on Advanced Information Systems Engineering (CAiSE90), Kista, Sweden, 1990.

[Winslett, 1990] Winslett, M. *Updating Logical Databases*, Cambridge University Press, Cambridge, U.K., 1990.

[Wiederhold et al, 1975] Wiederhold G., Fries J.F., Weyl S., *Structured Organization of Clinical Databases*, in Proceedings of the NCC, AFIPS Press, Montvale, New Jersey, 1975.