A Rule-Oriented Formalism for

Active Temporal Databases

Babis Theodoulidis, Pericles Loucopoulos and Vassilis Kopanas

Information Systems Group
Department of Computation
UMIST
P.O. Box 88
Manchester M60 1QD
email: babis@uk.ac.umist.co.isg

Abstract

This paper introduces a declarative conceptual modelling formalism and a database transaction handling mechanism for active temporal database applications. The term ‘active temporal databases’ is used in this paper to refer to databases that support historical information and in addition, provide mechanisms to express and handle constraints, derivations and actions on the stored information.

The conceptual modelling languages are the Entity-Relationship-Time (ERT) model, which deals with the definition of the structural aspects of the application including time modelling and complex objects, and the Conceptual Rule Language (CRL) which deals with the definition of constraints, derivations and actions expressed on the ERT model. At the database operation level the functionality expressed by the ERT model and the CRL language is realised by the Database Rule Language (DRL) which uses trigger-condition-action rules as a generic mechanism for the specification of procedures which are executed automatically when certain conditions arise.

1. Introduction

In recent years there has been an increasing demand for enhanced functionality of database systems that deal with applications beyond the traditional data processing type. It has also been argued that the development of the next generation of information systems will require the adoption of approaches which provide a closer alignment between business policy and system operation [19,34]. A number of issues arising from these requirements are addressed
within the TEMPORA project [20] which provides the framework for the work reported in this paper. The need for enhanced system functionality is addressed through the use of a conceptual modelling formalism which caters for: the modelling of business rules, the modelling of time and the modelling of complex objects. This formalism is supported at the database level by an extension of the relational model with temporal semantics and an execution mechanism that provides active database functionality.

One of the major advantages of modelling business rules is the separation of business policy from its implementation in application programs with the effect that any changes to business policy can be accommodated in an information system in a more straightforward manner than if the policy was merely represented in programming code. The effect of separating business policy from programming code was clearly demonstrated in an application which was concerned with 8 business policy rules for implementing a payroll system, required 3,500 lines of COBOL program to implement it [5]. The simplicity of the problem at the conceptual level (8 business rules) was counterbalanced by complex and voluminous code much of which had little to do with the problem in hand but rather with its efficient implementation. Many of the rate determination rules were implemented by the order of the program statements. Furthermore, few people could check the correctness of the implemented policy because few people with the knowledge of the rate calculation system were able to understand the implementation.

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 [37]. 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 [3,6,7,11,13,21]. Research interest in the time modelling area has increased dramatically over the past decade as shown by published bibliographies [23] and comparative studies [33].

The need to accommodate complex objects [2] arises from the fact that many applications have to deal with the modelling and management of objects of arbitrary complexity and to reason about them either in 'whole' or in a 'decomposed' form. Traditional data models fail to deal with this requirement. Extensions to the relational model include new types of attributes [12] and the relaxation of the first normal form constraint [1]. In both cases modelling of complex objects is carried out from a machine rather than a user-oriented perspective. The addition of time modelling of complex objects provides an extra level of functionality and semantic richness of the resultant conceptual schema.

This paper introduces a declarative design formalism for active temporal database

---

1 The TEMPORA project has been partly funded by the Commission of the European Communities under the ESPRIT R&D programme. The TEMPORA project is a collaborative project between: BIM, Belgium; Hitec, Greece; Imperial College, UK; LPA, UK; SINTEF, Norway; SISU, Sweden; University of Liege, Belgium and UMIST, UK. SISU is sponsored by the National Swedish Board for Technical Development (STU), ERICSSON and Swedish Telecomm.
applications. Modelling of business rules is carried out in terms of the Conceptual Rule Language (CRL) which can be used to define constraints or derivations on objects or conditions under which certain actions may take place. Modelling of objects - including time modelling and complex objects - is carried out using the Entity Relationship Time (ERT) model. The ERT and CRL are detailed in section 2. An abstract view of the relationship between the conceptual level components is shown in figure 1.

![Figure 1: Relationship of Conceptual Components](image)

The expressive power of the proposed formalism would be little use without a suitable database execution mechanism capable of supporting the functionality offered by the formalism. At the database level, the feasibility of this requirement is realised endowing a traditional DBMS with active support. An active database system can react automatically when certain conditions arise, without necessarily a user intervention. Active database systems are currently the focus of many research groups and more importantly, their potential and applicability has been recognised by vendors of database products who attempt to enhance the state-of-the-art database technology with active features. For example, declarative rules for database manipulation have been proposed in [22,24,29,38] and the concept of trigger has been included in commercial DBMS products [30]. In this paper, active database support is handled through the definition of a Database Rule Language (DRL) which uses trigger-condition-action rules as a generic mechanism for the specification of procedures which are executed automatically when certain conditions arise. The description of the DRL is given in section 3.

2. The Conceptual Modelling Languages

2.1 The Entity-Relationship-Time Model

2.1.1 Basic Concepts and Externals

The orientation of the ERT model is the Entity-Relationship formalism which makes a clear distinction between objects and relationships. 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 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. These two named roles are called relationship involvements and for completeness reasons, they are always required in an ERT schema. By using relationship involvements we have the possibility to express each relationship with two sentences which are syntactically different but semantically equivalent.

Time is introduced in ERT as a distinguished class called time period class. More specifically, each time varying simple entity class or complex entity class and each time varying relationship class is timestamped with a time period class. That is, a time period is assigned to every time varying piece of information that exists in an ERT schema. 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 simple entity class or complex 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.

Besides the objects for which history is kept, another type of object, called event is also supported. These are objects that prevail for only one time unit and thus, the notion of history does not apply to them. Alternatively, one can say that these objects become history as soon as they occur. Events are denoted by defining the duration of their timestamp to be one time unit.

As a consequence of the adopted timestamping semantics, only the event time is modelled in ERT; i.e. the time that a particular piece of information models reality. At a first glance this might seem to be restrictive in the sense that the captured information is not semantically as rich. However, this assumption is considered to be necessary in order to keep the proposed approach manageable and to permit computational attractive algorithms for reasoning about time.

For each timestamp its granularity should be defined. The granularities however, should be carefully chosen as they affect the relative ordering of events stored in the database. For example, if DATE is the granularity of the SHIPMENT object, then two shipments received on the same date will be treated to have occurred at the same time even if their exact time of arrival differs.

Another distinguished class that is introduced in ERT is that of a complex object. The distinction between simple and complex objects is that simple objects are irreducible in the sense they cannot be decomposed into other objects and thus, they are capable of independent existence whereas a complex object is composed of two or more objects and thus, its existence might depend on the existence of its component objects. The
relationship between a complex object and its component objects is modelled through the use of the IS_PART_OF relationship.

The ERT model accommodates explicitly generalisation/specialisation hierarchies. This is done through a distinguished ISA relationship which has the usual set theoretic semantics. Furthermore, for each relationship involvement, a user supplied constraint rule must be defined which restricts the number of times an entity or value can participate in this involvement. This constraint is called cardinality constraint and it is applied to the instances of this relationship involvement by restricting its population.

Each of the simple entity classes and user defined relationship classes in an ERT schema can be specified as derived. This implies that its instances are not stored by default but they can be obtained dynamically i.e. when needel, using the derivation rules. For each such derivable component, there is exactly one corresponding derivation rule which defines the members of this entity class or the instances of this relationship class at any time. In addition, if the derivable component is not timestamped then the corresponding derivation rule instantiates this component at all times whereas if this component is time varying then the corresponding derivation rule obtains instances of this class together with its existence period or validity period. In figure 2 an example ERT schema is given.

Figure 2: An example ERT schema

As shown in this schema, entity classes e.g., EMPLOYEE are represented
using rectangles with the addition of a 'time box' when they are time varying and derived entity classes e.g., Productive Employee are represented as dashed rectangles. Value classes e.g., Name are also represented with rectangles but with a small black triangle at the bottom right corner. Complex entity classes e.g., CAR and complex value classes e.g., ADDRESS are represented using double rectangles. Relationship classes are represented using a small filled rectangle (e.g. the relationship class between MANAGER and CAR) whereas derived relationship classes are represented using a non filled dashed rectangle. In addition, relationship involvements and cardinality constraints are specified for each relationship class. The interpretation of these is for example “a MANAGER has one CAR and a CAR belongs to only one MANAGER” (the diagram shows minimum and maximum for each relationship involvement).

The notation of the ISA hierarchies e.g., Productive Employee ISA EMPLOYEE is also given in figure 2. 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. It is assumed of course, that these constraints are applicable to hierarchies under the same specialisation criterion. These constraints are defined as follows:

- The **partial ISA constraint** states that there are members in the parent or generalised entity class that do not belong in any of its entity subclasses. On the other hand, the **total ISA constraint** states that there no members in the parent or generalised entity class that do not belong in any of its entity subclasses.

- The **overlapping ISA constraint** states that the subclasses of a given parent class under the same specialisation criterion are allowed to have common entities whereas the **disjoint ISA constraint** states that the subclasses of a given parent class under the same specialisation criterion are not allowed to have common entities.

The first of these constraints refers to the relationship between the parent class or generalised class and the child class(es) or specialised class(es). The second constraint refers to the relationship between child classes. 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.

### 2.1.2 Time Semantics

In the approach described in this paper, time is introduced as a distinguished entity class. For example, each entity class can be timestamped in order to indicate that its history is of importance to the particular application. The same argument applies also to user defined relationship classes and IS_PART_OF relationships.

The ‘time periods’ approach was chosen as the most primitive temporal notion because it satisfies the following requirements [16,35,36]:

...
• Period representation allows for imprecision and uncertainty of information. For example, modelling that the activity of eating precedes the activity of drinking coffee can be easily represented with the temporal relation before between the two validity periods [4]. If one tries, however, to model this requirement by using the line of dates then a number of problems will arise since the exact start and ending times of the two activities are not known.

• Period representation allows one to vary the grain of reasoning. For example, one can at the same time reason about turtle movements in days and main memory access times in nanoseconds.

The formal framework employed as the temporal reasoning mechanism is that of Interval Calculus proposed in [4] and which was later refined in [17] but with the addition of a formal calendar system in order to provide for the modelling and reasoning about the usual calendar periods. In figure 3, the semantics of the adopted time model is defined using ERT notation.

The modelling of information using time periods takes place as follows. First, each time varying object (entity or relationship) of ERT is assigned an instance of the built-in class SymbolPeriod. Instances of this class are system-generated unique identifiers of time periods e.g. SP1, SP2, etc. Members of this class can relate to each other by one of the thirteen temporal relations between periods [4]. Instances of the SymbolPeriod class may be displayed in an ER T schema. If they are not included then a simple T in the time box indicates that the corresponding object is timevarying.

The two subclasses of the Time Period class are disjoint as indicated in figure 3. This is because symbol periods are used to model relative time information while calendar periods model absolute time information. Thus, both views are accommodated.

In figure 3, the symbol τ represents a temporal relationship and the symbol τι
its inverse. In addition, time periods start and end on a *tick* and also have a duration expressed in ticks. A tick is defined as smallest unit of time that is permitted to be referenced and it is usually the time unit *second*.

The *CalendarPeriod* class has as instances all the conventional Gregorian calendar periods e.g., 10/3/1989, 21/6/1963, etc. Members of this class are also related to each other and to members of the Symbol Period class with one or more of the time period comparison predicates. The temporal relationships between calendar periods follow a formal calendar system [31,32] which is based on the work reported by Clifford and Rao in [10] with the addition of the calendar unit 'week'. This was considered to be necessary since there is often reference to this unit depending on the particular application domain.

Any desirable calendar time unit can be defined as a combination of already defined calendar units. Thus, expressions like END_OF_MONTH and NEXT_FORTNIGHT are easily defined. The usual operators are provided including set operators and comparators. These are distinguished to those that are applied to elements of the same domain and those that are applied to elements of different domains [10]. Additional operators like the time period comparison predicates are also provided together with functions that transform elements of one domain onto another. Note however, that the reasoning always takes place at the lower level of the ones involved.

Other notions of time such as *duration* and *periodic time* are also represented directly in the proposed formalism in addition to the above specified ones. As a consequence, the expressive power of the proposed formalism is increased and so does its usability. These notions of time are expressed in the Conceptual Rule Language as constraints upon the structural components and also as constraints on the behaviour of procedures.

The definition of the duration class is shown in figure 4. Members of this class are simple durations expressed in any abstraction level. Each duration consists of an amount of calendar time units expressed using real numbers and it is uniquely identified by the combination of its amount and abstraction level. For example, the duration '1.5 year' is a valid duration according to this definition.

![Metamodel of the duration class](image)

Figure 4: Metamodle of the duration class

The periodic time class is defined in figure 5.
As seen in this figure, a periodic time has a base which is a calendar period, a duration and also it can be restricted by a two calendar periods which restrict the set of values for this periodic time. For example, the expression 'first week of each month during next year' is a valid definition of a periodic time according to the above definition. In this case the calendar period corresponds to '1-7 days', the duration corresponds to '1 month' and the restricting calendar period is the next year corresponding to [1/1/1991, 31/12/1991]. A periodic time is uniquely identified by the combination of its base and its duration.

2.1.3 Complex Object Semantics

The basic motivation for the inclusion of the complex entity class and complex value class is from a methodological perspective, i.e. providing a designer with the ability to abstract away detail which in a particular situation is of no interest. No distinction is made between aggregation and grouping but rather a general composition mechanism for complex objects is introduced which also involves relationship classes.

Graphically, composition is shown by surrounding the components with a rectangle representing the composite object class. The notation of a complex object in ERT is shown in figure 6. The complex value class ADDRESS and the complex entity class CAR of figure 2 may be viewed at a more detailed level as shown in figures 6 and figure 7 respectively.
arranged substructures. Each directly subordinate component entity class is part_of-related to the complex entity class border so that the relationship between the composite object and its components will be completely defined. Whether the HasComponent involvement is one of aggregation or grouping, it can be shown by means of the normal cardinality constraints. That is, if its cardinality is (0,1) or (1,1), the component is aggregate whereas if its cardinality is (0,N) or (1,N), the component is a set.

![Diagram](image)

Figure 7: The complex entity class CAR in more detail

Most conceptual modelling formalisms which include complex objects [14,18,27], model only physical part hierarchies i.e, hierarchies in which an object cannot be part of more than one object at the same time. In the ERT model, this notion is extended in order to be able to model also logical part hierarchies where the same component can be part of more than one complex object.

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). That is, the following kinds of IS_PART_OF variations [15] are accommodated:

- dependent exclusive composite reference
- independent exclusive composite reference
- dependent shared composite reference
- independent shared composite reference

Note that no specific notation is introduced for these constraints. Their interpretation comes from the cardinality constraints of the IS_PART_OF relationship. That is, assume that the cardinality of the IS_PART_OF relationship is \((\alpha,\beta)\). Then, \(\alpha=0\) implies
non dependency, $\alpha \neq 0$ implies dependency, $\beta = 1$ implies exclusivity while $\beta \neq 1$ implies shareness.

Complex object classes and IS_PART_OF relationship classes can be time-varying as shown in the above examples. This is discussed in detail later where the interaction of complex objects with time is elaborated. In particular, the way complex object hierarchies evolve over time and a number of constraints should always be valid during this process.

### 2.2 The Conceptual Rule Language

The CRL language is concerned with constraints placed upon the elements of ERT, with the derivation of new information based on existing information and with the invocation of procedures that denote execution semantics of transactions. Within the CRL formalism, different rule types are distinguished in order to achieve orthogonality of concepts and more assistance in the rule elicitation process. In addition, a textual layout with natural language semantics was adopted for the CRL language in order to increase its understandability and usability.

The rule classification schema is shown in figure 8. As shown in this figure, the following different types of rules are distinguished:

- **Constraint rules** which are concerned with the integrity of the ERT components. They are further subdivided to static constraint rules which are expressions that must hold in every valid state of an ERT database and transition constraint rules which are expressions that define valid state transitions in an ERT database. An example of a static constraint rule might be 'The number of employees working in a department must be less than 100 at all times'. An example of a transition constraint rule might be 'The salary of an employee must never decrease'.

- **Derivation rules** which are expressions that define the derived components of the ERT model in terms of other ERT components including derived components. There must be exactly one derivation rule for each such component. As the constraint rules, derivations rules are also subdivided to static derivation rules and transition derivation rules depending on whether the derived ERT component is timestamped or not. An example of a static derivation rule might be 'A supplier is the cheapest supplier for a particular product if his offer for this product has the minimum price'. An example of a transition derivation rule might be 'A customer is the best customer of this month if the total amount of his orders placed this month is the maximum'.

- **Action rules** which are concerned with the invocation of procedures. In particular, action rules express the conditions under which procedures are considered fireable i.e., a set of triggering conditions and/or a set of preconditions that must be satisfied prior to their execution. An example of an action rule might be 'When the stock of a product falls below the reorder quantity level specified for this product then execute
the reorder procedure immediately'.

![Figure 8: Classification of CRL rules](image)

The inclusion of an explicit time component in the proposed formalism allows for the expression of a wider class of constraint rules in the same language, something that is not possible in a first order language without some extra apparatus. As specified previously, 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 tick.

More specifically, static constraint rules apply to every state of the database and thus, they are time independent. In fact, they represent definitions of conditions which must hold between different classes (entity, value or relationship classes) in any individual state.

The purpose of a static constraint rule is to restrict each valid state of one or more items of data and it can be said to hold (or not hold) simply on the basis of the extension of the database with respect to a single state. These rules are also called extensional constraints, functional dependency rules and multivalued dependency rules and their mapping to first order logic formulas is well defined [25,26].

On the other side, transition constraint rules place restrictions on two or more states of the database by specifying valid state progressions. This type of rules is possible to express directly in the context of the proposed formalism because of the explicit modelling of the evolution of data. Each transition constraint rule is said to hold (or not hold) only by examining at least two states of the database. These rules are also called intensional constraints [39], dynamic constraints and constraints upon update operations [9,28,40].

As stated already, static constraint rules are used to describe each permissible state of the ERT and thus, they are logical restrictions on the data. A rule in this category always refers to a property of an entity class or a value class or a relationship class and it can be either true or false. The description of this property is done in terms of imperative or conditional statements which are called state conditions and refer to classes or instances of classes. In fact, static constraint rules preserve the integrity of the database by restricting the operations that can be applied in any individual state and must be true at any time period. In particular, the following types of static constraint rules are distinguished:
• **On entity classes.** These static constraint rules are concerned with restrictions on the number of instances of a particular entity class or the way that the instances of a specific entity class may be identified.

• **On value classes.** These static constraint rules are concerned with restrictions imposed on the domain of values of a particular value class.

• **On relationship classes.** These static constraint rules are concerned with restrictions on the number of instances of a single entity class that are involved in different relationships and also, with restrictions between sets of instances of the same entity class that are involved in one or more relationships.

The basic structure for all rules expressed in CRL is given in the following BNF definition, where the expressions in bold brackets are optional. Any free variables that appear in the rule have implicit universal quantification.

\[
\text{CRL\_rule ::= } [ [ \text{WHEN} \ <\text{trigger\_exp}> ] \ [ \text{IF} \ <\text{cond\_exp}> ] \ \text{THEN} ] \ <\text{exp}>
\]

This leads to there being four valid variants of the basic CRL rule, listed here with their corresponding semantics.

• \(<\text{exp}>\)  
  \(<\text{exp}>\) must always hold.

• \(<\text{IF} \ <\text{cond\_exp}> \ \text{THEN} \ <\text{exp}>\)  
  \(<\text{exp}>\) must hold whenever \(<\text{cond\_exp}>\) holds.

• \(<\text{WHEN} \ <\text{trigger\_exp}> \ \text{THEN} \ <\text{exp}>\)  
  \(<\text{exp}>\) must hold when \(<\text{trigger\_exp}>\) has just begun to hold.

• \(<\text{WHEN} \ <\text{trigger\_exp}> \ \text{IF} \ <\text{cond\_exp}> \ \text{THEN} \ <\text{exp}>\)  
  \(<\text{exp}>\) must hold if \(<\text{cond\_exp}>\) holds, and \(<\text{trigger\_exp}>\) has just begun to hold.

To access the entities and values in the ERT model, a single general structure is used, defined by the BNF expression below, with the optional repeating sections in bold braces. Naming an entity or value class causes the access expression to hold for each instance of the class, and by enclosing a variable in parenthesis after the name to give the predicate form bindings of the variable can be obtained to each instance found. Enclosing a list of relationship names with other entities or values enables us to qualify our selection of instances by stating that the particular instance must be related to an instance of the other entity or value.

\[
\text{ERL\_data\_access ::= } <\text{entity/value name}> [ (<\text{variable}>)]
\]
3. The Database Rule Language

3.1 Database Transitions

In order to facilitate the mapping between the conceptual level and the run-time environment, the database rules syntax is as close as possible to the syntactic constructs used in the design level. Also, at this version, the DRL rule expressions provided follow closely the syntax of Prolog in order to facilitate its implementation. User-friendliness for the database level rule syntax is considered to be of low importance at this stage of development because database level rules are meant to be automatically generated by a mapper and thus, it is invisible for the user or the systems analyst. However, as part of future work, extensions of the syntax of DRL will be considered as well as its interfacing with SQL.

The expressions of DRL have the following format:

```
rule( Rule_name, Rule_priority,
     event( Event_expression ),
     condition( [ Conditions_list ] ),
     action( [ Actions_list ] ).
```

In the above expression `Rule_name` is an identifier for the rule. `Rule_priority` is a measure of significance for the rule taken from a predefined priorities interval. `Event_expression` is an expression specifying under which circumstances the rule can be activated. This expression can be either a simple event, a conjunction of simple events or a disjunction of simple events. A simple event is either a database modification (insert, delete, update), or an external signal (user signal, clock signal).

`Conditions_list` is a Boolean expression having the syntax of a valid Prolog clause body, containing both built-in and user-defined predicates. When the Conditions_list for an active rule is evaluated and found 'true', the rule is applicable and will be fired.

`Actions_list` is a list of database operations, external-procedure calls or the operation 'abort'. Database operations are set oriented insertions, deletions, updates. The target tuples for deletions and updates are specified inside the operation by an expression `Locate_expression` which has the same syntax as Conditions_list that was described above. In figure 9, the simple rule shown aborts a transaction when insertion is attempted to a subtype (single_order_customer) and the inserted item(s) do not exist in the supertype (customer).

In figure 10, the rule shown is activated by a deletion to a supertype (customer) and propagates the deletions to the subtype (single_order_customer).

The syntax of the DRL expressions follows closely the CRL action rules syntax. That is, both have a triggering, a precondition and an action part. In addition, by keeping a Prolog-based syntax, it is possible to easily extend it with temporal expressions and even
with natural language expressions.

```
rule( r1, 20,
  event((insert, single_order_customer)),
  condition([ inserted(L),
    member((single_order_customer,L2), L),
    for_all_members((_,[X]),L2),
    not_exists(customer(_,X,_,_)),
    action([abort])).
```

**Figure 9:** A rule that causes transaction abort

Before discussing the rule execution mechanism, it is necessary to introduce the adopted database state transition semantics. For this, let us consider a transition $T_{k+1}$ from database state $S_k$ to database state $S_{k+1}$ that is caused by the application of the actions set $A_{k+1}$ which is the set of operations found inside the action part of applicable rule(s) or simply externally submitted operations (see Figure 11). $T_k$ is the set of active rules just after state $S_k$ while $E_{k+1}$ is the net effect of transition $T_{k+1}$.

```
rule( r2, 20,
  event((delete, customer)),
  condition([ true]),
  action([my_delete(single_order_customer,[X],
    (deleted(L),
    member((customer,L1),L),
    member((_,[X,_,_]),L1))))
```

**Figure 10:** A rule that propagates deletions.

Note that **rule condition activation, evaluation occurs only before a transition starts or after a transition has completed.** Figure 13 shows the possible types of operations that can cause a database state transition.

![Database state transition diagram](image-url)

**Figure 11:** A database state transition.

As it is shown in figure 12, **case 1** considers actions consisted of a single operation for a single instantiation of a rule's condition. Similarly, **case 2** considers actions consisted of a single operation for a set of instantiations of a rule's condition. **Case 3**, considers
actions consisted of a set of operations for a single instantiation of a rule's condition. Case 4, finally, considers actions consisted of a set of operations for a set of instantiations of a rule's condition.

<table>
<thead>
<tr>
<th>Single instantiation</th>
<th>Set-of instantiations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single operation</td>
<td>CASE 1</td>
</tr>
<tr>
<td>Set-of operations</td>
<td>CASE 3</td>
</tr>
</tbody>
</table>

Figure 12: Database transition semantics cases

For cases in the same row of figure 12, 'set-of-instantiations' type of cases (column 2) are more preferable than 'single-instantiation' type of cases (column 1). This is because, these cases exploit set-oriented processing and optimisations offered by relational DBMSs. Consequently, case 2 is more preferable than case 1 and case 4 is more preferable than case 3.

Similarly, considering same column cases in figure 12, 'set-of-operations' cases (row 2) are more preferable than 'single-operation' cases (row 1). This is because 'set-of-operations' semantics (row 2 cases), allow us, optionally, to group database operations and apply them as a whole (atomic unit). This way, the next triggering cycle will be invoked by the net effect of the operation block application. Using this facility it is possible to avoid triggering after each individual operation inside the block. Consequently, case 3 is more preferable than case 1 and case 4 is more preferable than case 2.

Based on the above discussion, a mixed case 2, case 4 transition semantics is followed when the actions set consists of externally submitted operations, while case 4 transition semantics is used for action sets consisting of operations from action parts of applicable rules.

Of particular importance is the semantics of a rule as part of a rule set. To exemplify this let us consider a state transition $T^k$ which results in a database state $S_k$ and a set $T_k$ of triggered rules. At first, these rules are ordered according to some priority measure. Then, two groups of rules can be distinguished:

i) $Group \ A \ (Ga)$ which contains all rules in $T_k$ having the same priority value, maximum priority inside $T_k$.

ii) $Group \ B \ (Gb)$ which contains all the rules in $T_k$ which are not inside Group A.

Rules belonging to Group A are considered of equivalent importance. Any rule from Group A is considered much more important than any rule from Group B. For rules in Group A, all preconditions are evaluated first and then the actions set is formed using the
applicable ones. The actions set is tested for consistency and then it is applied (action order is insignificant). A transition, say $T_{k+1}$ occurs giving state $S_{k+1}$.

Group B rules from set $T_k$ on the other side, are carried forward inside $T_{k+1}$ together with the newly triggered rules $T_{\text{new}}$. $T_{k+1}$ is given by the formula:

$$T_{k+1} = (G_b)_k \cup [T_{\text{new}} - ( (G_b)_k \cap T_{\text{new}} ) ]$$  (6)

Again $(G_a)_{k+1}$, $(G_b)_{k+1}$ is formed and the algorithm proceeds in the same way until $T_n = \emptyset$ where $S_n$ is the most recent state.

This execution mechanism implies the following declarative semantics for members of the rule-set: "When the triggering event becomes true and the rule under consideration has become member of a Group A rule set, then, for all instantiations of $X$ such that condition $C(X)$ becomes true, the action part $A(X)$ also must be true".

By adopting the above semantics, rules of equivalent importance are handled similarly. That is, preconditions for all $(G_a)_k$ rules are evaluated on the same state $S_k$, while the actions set corresponding to $(G_a)_k$ rules is applied without being delayed by precondition evaluation inside $(G_b)_k$. High priority rules, on the other side, triggered by transition $T_{k+1}$ are handled before low priority rules found in $T_k$. More accurately, $(G_a)_{k+1}$ is tested for applicability before any $R_i: R_i \notin ( (G_b)_k - (G_a)_{k+1} )$. This way, the concept of prioritisation and privileged handling of rules is introduced, while at the same time rules of similar importance can be handled properly.

By specifying a method finally, to determine the 'width' of $(G_a)_k$ one can have an easy method to affect the behaviour of the rule manager either manually or automatically through meta rules. This way, the flexibility of a rule manager with easily modifiable policy is provided.

According to the execution model, the system receives input from its environment through an external queue. Any item inside this queue can be of any of the following types:

1. Individual items (insertions, deletions, updates, signals). These are submitted by an external agent (user, application programme, system clock).

2. Operation blocks submitted by an external agent (user or application programme).

In order to process external queue items the system uses an internal queue where any external queue item that has to be processed is placed (only one item at a time). Depending on the type of the item under consideration different execution mechanism characteristics apply, reflecting the semantic differentiation between items of the above mentioned types. Rules on the other side are considered for activation either in immediate mode (mode-i) or in deferred mode (mode-d).

The following discussion presents the run-time rule manipulation as well as the resulting execution mechanism.

In figure 13, $S_{kj}$ represents database state $j$ inside transaction branch $k$. $O_{k1}$ is an
internal queue item that initiates transaction branch \( k \). \( \text{Ob}_{kj} \) is an actions set containing operations taken from the action part of applicable Group A rules. Branch \( k \) starts with an item \( O_{k1} \) and terminates when no more active rules exist or can be identified. Each arrow between adjacent states \( S_{k-1}, S_k \) represents a transition \( T_k \) with net effect \( E_k \).

Here, \( T_{k1} \) follow case 2 of transition semantics while \( T_{kw}, w>1 \) follow case 4 transition semantics. A transaction branch \( k \) (see Figure 13) starts with an independent operation or signal \( O_{k1} \) (not member of an operations block) submitted by an external agent (user, application programme, system clock). Inside branch \( k \) then, all rules (mode-i, mode-d) are enabled. After each transition, caused by \( O_{k1} \) or an \( \text{Ob}_{kj}, j=2, 3, ...n(k) \), all rules are tested for activation by evaluating their transition predicate (WHEN part) against the net effect of \( O_{k1} \) or \( \text{Ob}_{kj} \) respectively.

### 3.2 Rollback semantics

If inside branch \( k \), transaction abort occurs (either explicit abort command or invalid database operation), the whole branch \( k \) is rollbacked, to state \( S_{k0} \).

If branch \( k \) execution is successful the final state is \( S_{k,n(k)} \). If on the other hand branch \( k \) execution is unsuccessful, the final state is \( S_{k0} \). In either case, the internal queue is empty and the system proceeds with the next item from the external queue (\( O_{(k+1)1} \)). Note here that user signals and clock signals can be easily accommodated to the above mentioned framework. Signals are used as valid \( O_{k1} \) items which when applied result in \( S_{k0}=S_{k1}, E_{k1}=[] \). Signals \( O_{k1} \) application however, is always a valid operation. Additionally signals \( O_{k1} \) cannot be rollbacked. External actions (e.g. ring alarm) finally, that occur inside \( \text{Ob}_{kl} \), are always valid operations and cannot be rollbacked. Their application results in \( S_{k(r-1)}=S_{kr}, E_{kr}=[] \).

For the case where external operations \( [O_{11}, O_{21}, ...O_{m1}] \) form an operations block the same symbol semantics as previously are used. In this case (see figure 14), \( O_{11}, O_{21}, ...O_{k1}, ...O_{m1} \) are members of an operations block submitted by an external agent (user, application programme).

We should note here that signals are not allowed inside an operations block. The first operation inside the block is \( O_{11} \) and the last is \( O_{m1} \). Just before \( O_{11} \) is applied (see figure 14), (the first branch for the operations block starts), all mode-d rules are disabled. These
rules remain disabled until $\text{Ob}_{m,n(m)}$ is applied completely. After each transition, (caused either by an $O_{j1}$ or an $\text{Ob}_{ji}$, $j=1,2,...,m$, $i=1,2,...,n(j)$ ), only mode-$i$ rules are tested for activation by evaluating their transition predicate against the net effect of $O_{j1}$ or $\text{Ob}_{ji}$ respectively.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure14.png}
\caption{Transaction branches caused by an externally submitted operations block.}
\end{figure}

In state $S_{m,n(m)}$ no mode-$i$ rules remain active and no more mode-$i$ rules can be activated. We then say that operations block $[O_{11}, O_{21},...,O_{m1}]$ has been fully applied. It is exactly this point of time when we enable mode-$d$ rules. All mode-$d$ rules are tested for activation by evaluating their transition predicate against the net effect of the complete operations block application (all $m$ branches up to and including branch $m$ part a). Action part application for mode-$d$ rules results in transition $T_{m,n(m)+1}$ and final state $S_{m,n(m)+1}$. Triggering branch $m$ part b then evolves up to the final state $S_{m,w(m)}$ where no rule is active or can be activated. We say then, that the operations block with all its consequences has been applied. Let us consider an intermediate state $S_{m,n(m)+r}$ where $n(m)+r < w(m)$. This state has occurred as a result of applying action set $O_{m,n(m)+r}$ with net effect $E_{m,n(m)+r}$. In this state, set $P_{m,n(m)+r}$ and consequently the corresponding $Ga$ may contain two types of rules:

\textbf{Type A} Mode-$d$ rules activated in state $S_{m,n(m)}$ but not considered for applicability and firing up to state $S_{m,n(m)+r}$. 

Type B  Mode-d or mode-i rules activated in state $S_{m,y}$, $n(m)+1 \leq y \leq n(m)+r$ but not considered for applicability and firing up to state $S_{m,n(m)+r}$.

The applicability of rules from $G_{m,n(n)+r}$ is tested according to the following rules:

- If the rule is of type A, its condition is evaluated against $E_b \circ E_{m,n(m)+1} \cdots \circ E_{m,n(m)+r}$ transition tables and using also state $S_{m,n(m)+r}$. Here $E_b$ is the net effect of the operations block between states $S_{10}$ and $S_{m,n(m)}$ while '$\circ$' is the net effect accumulation operator.

- If the rule is of type B or C, its condition is evaluated against $(E_{m,(y+1)} \cdots \circ E_{m,n(m)+r})$ transition tables and using also state $S_{m,n(m)+r}$. Again, '$\circ$' is the net effect accumulation operator.

The rollback semantics are as follows:

CASE 2a. If inside transaction branch $k$ of the operations block, transaction abort occurs (either explicit 'abort' command or invalid database operation), the whole branch $k$ is rollbacked together with branches 1, 2, ..., $k-1$. The final state is $S_{10}$. Additionally, any remaining part of the operations block (operations $O_{(k+1)1}$, $O_{(k+2)1}$, ..., $O_{m1}$) are discarded from the internal queue.

CASE 2b. If all $m$ branches (up to branch $m$ part a) terminate successfully but inside transaction branch $m$ part b of the operations block, transaction abort occurs (either explicit 'abort' command or invalid database operation), the whole branch $m$ is rolled back together with all previous branches (1, 2, ..., $m-1$) of the same operations block. The final state is $S_{10}$.

In summary, if operations block execution is successful, we get to state $S_{m,w(m)}=S_{(m+1)0}$. If on the other hand, operations block execution is unsuccessful we get to state $S_{10}$. In both cases the internal queue is empty and the system proceeds with the next item from the external queue.

4. Conclusions and Future Work

Contemporary approaches to information system development, whilst attempting to improve the management of developing such systems through the use of software engineering methods and CASE, have paid little attention to the requirements for effective system evolution and for using information systems for effecting changes at the strategic level of organisations. The premise behind the work reported in this paper is that information system development is about formalising and documenting knowledge about the universe of discourse and this knowledge should be represented explicitly and independently to the way that it is implemented in data structures and algorithms thus leading to a more efficient way of developing and maintaining software. In particular, this approach seeks to separate out and explicitly maintain throughout the software lifecycle, the notion of policy,
as described by *constraint*, *derivation* and *action* rules.

This paper seeks to demonstrate how a set of business rules may be interpreted by analysts in terms of the objects that exist in the organisation and their structural relationships (using the ERT model) and the rules that are expressed by references to these objects (using the CRL). Furthermore, the paper demonstrates how this conceptually oriented specification can be translated into an executable specification and outlines the major components of a rule manager capable of handling transactions at the run-time application level (using the DRL).

Current work relating to the issues discussed in this paper is concerned with the development of CASE tools to support the design process, developing mapping tools between the conceptual and executable levels, coupling a Prolog implementation of the DRL to the SYBASE DBMS and testing the feasibility of the paradigm on large scale industrial applications.

References


[17] ESPRIT P107-LOKI, A Logic Oriented Approach to Knowledge and Databases Supporting Natural Language User Interfaces, Institute of Computer Science, Research Center of Crete, Greece, March 1986.


