[9]

Integrating database technology, rule-based systems and temporal reasoning for effective information systems: the TEMPORA paradigm

P Loucopoulos,* P McBrien,† F Schumacker,‡ B Theodoulidis,*
V Kopanas* and B Wangler§

*Department of Computation, Manchester and †Department of Computing, Imperial College, London, UK, ‡Service d'Informatique, Universite de Liege, Institute Montefiore, Belgium and SISU, Kista, Sweden

Abstract. *Recent years have witnessed a growing realization that the development of large data-intensive, transaction-oriented information systems is becoming increasingly more difficult as user requirements become broader and more sophisticated. Contemporary approaches have been criticized for producing systems which are difficult to maintain and which provide little assistance in organizational developments. This paper introduces the TEMPORA paradigm, which is currently under development and which advocates a closer alignment between organizational policy and information system functionality. This viewpoint impacts on a number of critical issues related to the development process of information systems most notably in the nature of conceptual models, the discipline adopted for the development, the type of support provided by CASE tools and the run-time environment. The paper introduces the philosophy and architecture of the TEMPORA paradigm and describes the conceptual models, tools and run-time environment which render such an approach a feasible undertaking.*

Keywords: business rules, CASE, conceptual modelling, rule-based paradigm, temporal database.

INTRODUCTION

Much has been written about the benefits and problems of contemporary approaches to the development of large-scale information systems (Maddison, 1983; Olle *et al.*, 1983, 1986). Despite the undeniable benefits that these approaches have brought about, two unresolved.

Correspondence Pericles Loucopoulos, Department of Computing, UMIST, P.O. Box 88, Manchester, M60 1QD, UK.

First, there is very little explicit correspondence between business 'rules' and information systems. This has the effect that the information system can be neither easily examined to ascertain whether it is still aligned to business practice, nor used effectively to help develop new enterprise designs. Second, the issue of maintenance and system evolution continues to be a major problem in commercial software. A large part of this problem is the way that the factors (mostly business policy) that cause change are not explicitly maintained, and even worse, their representation in software is embedded together with programming code whose function is concerned with issues such as file accessing, input and output, sequencing of operations, data integrity, etc. A relatively simple example demonstrating this point reported in (Anderson *et al.*, 1986), deals with eight business policy rules for implementing a payroll system. The implementation of this business application required 3,500 lines of COBOL code. The simplicity of the problem, when expressed in business terms (eight rules), was counterbalanced by a complex and voluminous code, much of which had little to do with the actual problem in hand, but rather with its efficient implementation. The result was that business people could not check the correctness of the implemented policy as the people with the knowledge of the rate calculation system were unable, or had no inclination, to understand the implementation. The result, in this case, was that evolution of the system was difficult, as the implementation described the procedure to determine rates of pay rather than containing the policy of calculation at a level abstract enough to enable both developers and end users to reason about the business knowledge free of implementation considerations.

To address these problems, TEMPORA proposes that developers must be provided with a process which assists in modelling business policy and linking this policy to the software development process. TEMPORA advocates an approach which explicitly recognizes the role of business policy within an information system and visibly maintains this policy throughout the software development process, from requirements specifications through to an executable implementation. The need for such a paradigm has been recognized in the, now completed ESPRIT project RUBRIC (van Assche *et al.*, 1988; Loucopoulos, 1989). The TEMPORA project builds upon these early results and extends this work in two directions. The first direction is concerned with the use of a commercial DBMS as the underlying data management mechanism. The second direction is concerned with enhancing the paradigm with the explicit modelling of temporal aspects at both specification and application levels.

The TEMPORA architecture is shown in Fig. 1. An analyst/designer develops an information system specification using an interactive CASE tool environment which incorporates three tools, two of which correspond to the conceptual level in terms of the ERT (Entity–relationship–time) and process (external rule language) models and one which is concerned with the specification of application-related components. Later in this paper we discuss the conceptual models of TEMPORA. The ERT and process part of the specification are expressed in such a way as to enable one to reason at the business level (external level). Each component of the specification is mapped onto an execution layer (design level) which deals with data and execution mechanisms from a database schema perspective. We describe briefly the TEMPORA case tool environment which is used in order to develop a TEMPORA application. The actual mechanics of storing data, implementing constraints and executing temporal as well as non-

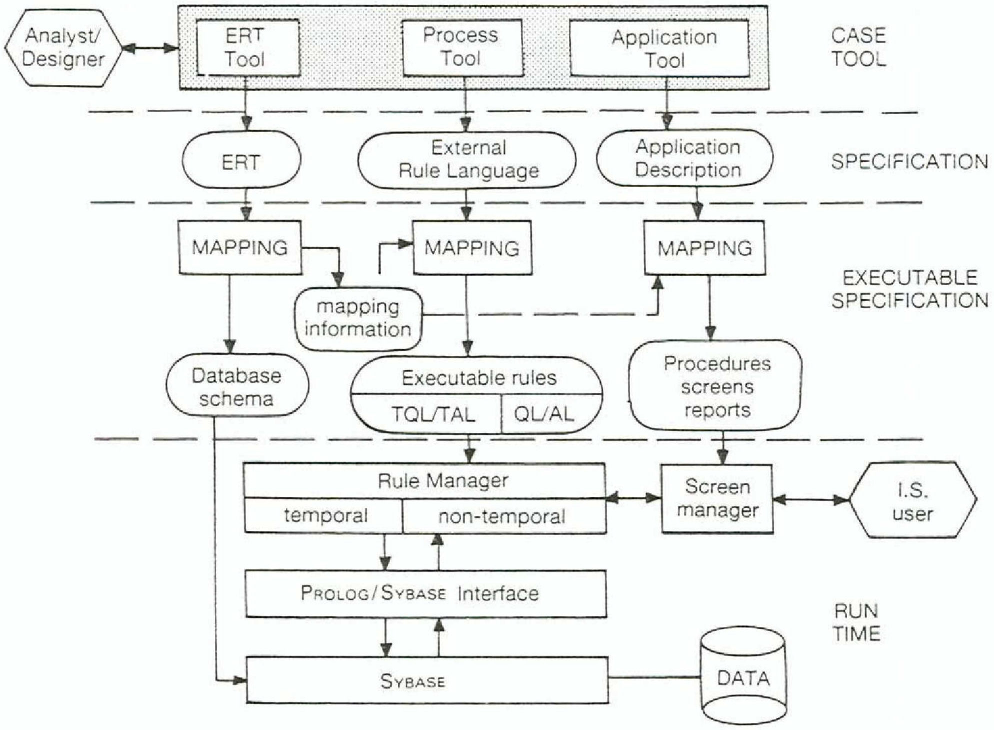


Figure 1. The TEMPORA architecture.

temporal rules is handled by the run-time environment which consists of the SYBASE DBMS and its interface to BIM_Prolog and an extension module referred to as the rule manager. The run-time environment will be described later.

CONCEPTUAL MODELS

The entity–relationship–time (ERT) model

Basic concepts

The components of ERT are defined as follows.

1 Entity is anything, concrete or abstract, uniquely identifiable and being of interest during a certain time period. Entity Class is the collection of all the entities to which a specific definition and common properties apply at a specific time period.

2 Relationship is any permanent or temporary association between two entities or between an entity and a value. Relationship Class is the collection of all the relationships to which a specific definition applies at a specific time period.

3 Value is a lexical object perceived individually, which is only of interest when it is associated with an entity. That is, values cannot exist in their own. Value Class is the proposition establishing a domain of values.

4 Time Period is a pair of time points expressed at the same abstraction level. Time Period Class is a collection of time periods.

5 Complex Object is a complex value or a complex entity. A complex entity is an abstraction (aggregation or grouping) of entities, relationships and values (complex or simple). A complex value is an abstraction (aggregation or grouping) of values (complex or simple). Complex Object Class is a collection of complex objects.

An entity or relationship can be derived. This implies that its value is not stored by default. For each such derivable component, there is a corresponding derivation rule which gives the members of this class or the values of this relationship at any time.

Time is introduced in the ERT model as a distinguished entity class. More specifically, each time-varying entity class and each time-varying relationship class is time stamped with a time period class. That is, a time period is assigned to every time-varying piece of information that exists in a schema. For example, for each entity class a time period is associated which represents the period of time during which an entity is modelled (existence period of an entity). The same argument applies also to relationships, i.e. each time-varying relationship is associated with a time period which represents the period during which the relationship is valid (validity period of a relationship).

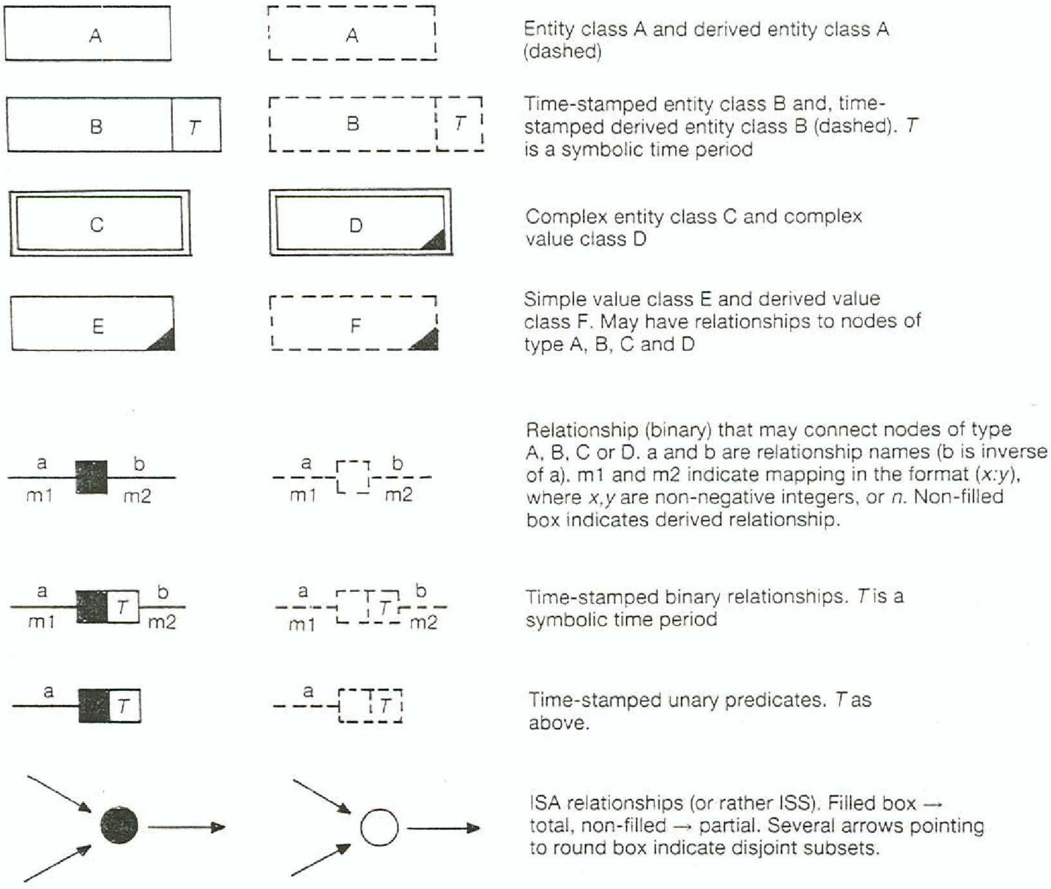
Figure 2 presents the notation for the ERT externals. Note that the graphical notation caters for the representation of some of the most common rules such as partial/total ISA relationships and cardinality constraints.

The ERT model accommodates explicitly generalization/specialization hierarchies. This is done through the ISA relationship which has the usual set-theoretic semantics. More specifically, it is assumed that two subclasses of the same entity class and under the same specialization criterion are always disjoint.

Cardinality constraints may be given to all relationships (including the IS_PART_OF relationship) and also to their respective inverse relationships. Note here that there is no separate notation for the IS_PART_OF relationships. However, their corresponding cardinality constraints are interpreted in a slightly different way. This is explained in more detail in the next section.

Complex objects

In general, complex objects can be viewed from at least two different perspectives (Batini, 1988): representational and methodological. The representational perspective focuses on the way entities in the real world should be represented in a conceptual schema and on.the way events in the real world are mapped onto operations on the corresponding objects. In contrast, if complex objects are not allowed, for example, in the relational model, then information about the object is distributed and operations on the object are transformed to a series of associated operations. The methodological perspective treats the complex object concept as a means of stepwise refinement for the schema and for hiding away details of the description. This in turn, implies that complex objects are merely treated as abbreviations that may be expanded when needed. In the context of the ERT model, complex objects are treated in terms of the methodological interpretation, i.e. they serve as a convenient abstraction mechanism.



Entity class A and derived entity class A (dashed)

Time-stamped entity class B and, time-stamped derived entity class B (dashed). *T* is a symbolic time period

Complex entity class C and complex value class D

Simple value class E and derived value class F. May have relationships to nodes of type A, B, C and D

Relationship (binary) that may connect nodes of type A, B, C or D. a and b are relationship names (b is inverse of a). m1 and m2 indicate mapping in the format (*x*:*y*), where *x*,*y* are non-negative integers, or *n*. Non-filled box indicates derived relationship.

Time-stamped binary relationships. *T* is a symbolic time period

Time-stamped unary predicates. *T* as above.

ISA relationships (or rather ISS). Filled box → total, non-filled → partial. Several arrows pointing to round box indicate disjoint subsets.

Figure 2. Graphical notation for the ERT model.

The notion of a complex object in ERT is shown in Fig. 3. The example ERT diagram shows a complex entity class CAR and a complex value class ADDRESS. Furthermore, the complex objects CAR and ADDRESS may be viewed at a more detailed level as shown in Fig. 4.

The components of a complex object comprise one or more hierarchically arranged substructures. Each directly subordinate component entity must be IS_PART_OF related to the complex object border so that the relationship between the composite object and its components will be completely defined. Whether the HAS_COMPONENT relationship is one of aggregation or grouping, 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.

Most conceptual modelling formalisms, which include complex objects (Lorie, 1983; Kim *et al.*, 1987; Rabitti *et al.*, 1988), 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 ERT, this notion is extended in order to be able to also model 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

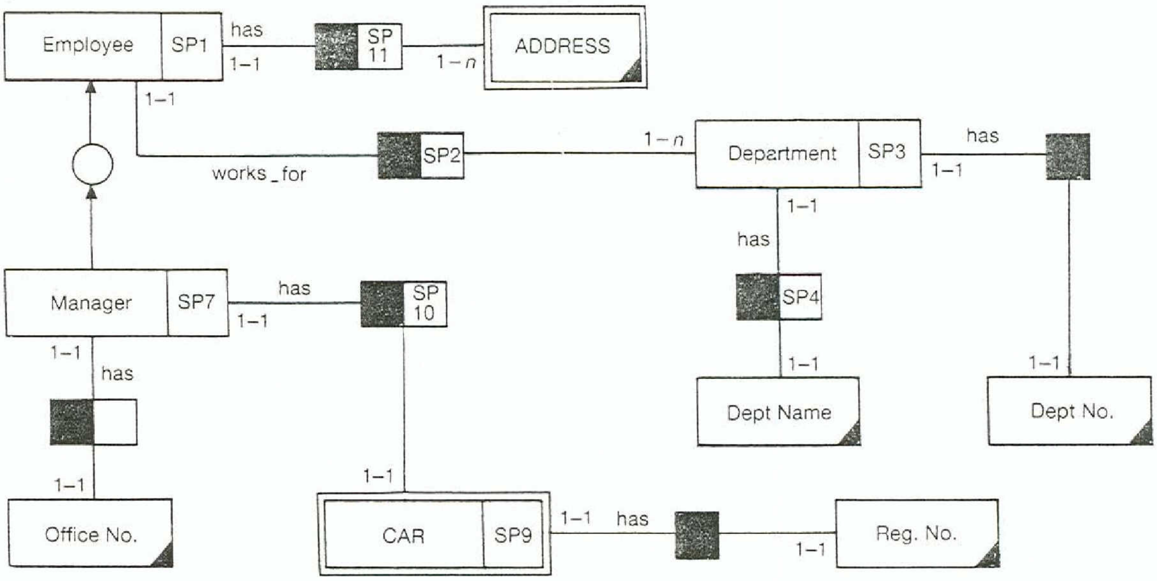


Figure 3. An ERT diagram.

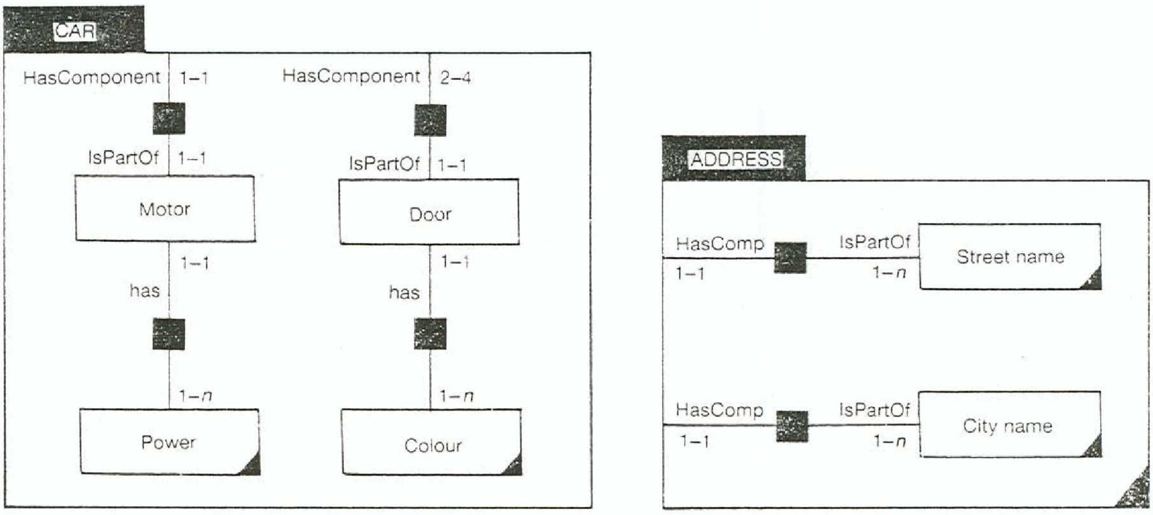


Figure 4. Components of complex objects CAR and ADDRESS.

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 (Kim, 1989) are accommodated:

(a) dependent exclusive composite reference,
(b) independent exclusive composite reference,
(c) dependent shared composite reference,
(d) 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, assuming that the cardinality of the IS_PART_OF relationships is (a, b) then, $a = 0$ implies non-dependency, $a \neq 0$ implies dependency, $b = 1$ implies exclusivity while $b \neq 1$ implies shareness.

Finally, the following rules concerning complex objects should be observed.

1 Complex values may only have other values as their components. In addition, the corresponding IS_PART_OF relationship will always have dependency semantics unless it takes part in another relationship.

2 Complex entities may have both entities and values as their components. Every component entity must be IS_PART_OF related to the complex entity.

3 Components, whether entities or values, may in turn be complex, thereby yielding a composition/decomposition hierarchy.

Time stamping semantics

The time period representation approach has been chosen because it satisfies the following requirements (Villain, 1982, 1986; Ladkin, 1987).

1 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 by placing the temporal relation 'before' between the two validity periods (Allen, 1983). If one tries, however, to model this requirement by using the line of dates, then a number of problems will arise as the exact start and ending times of the two activities are not known.

2 Period representation allows one to vary the grain of reasoning.

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 13 temporal relations between periods (Allen, 1983). The class CalendarPeriod has as instances, all the conventional calendric periods, e.g. 10/3/1989, 21/6/1963, etc. Members of this class are also related to each other and to symbol periods by one of the 13 temporal relations between time periods.

Figure 5 shows graphically the definition of these concepts using the ERT notion. The symbol τ represents a temporal relationship and the symbol τi its inverse. The fact that the two classes *SymbolPeriod* and *CalendarPeriod* are disjoint is also indicated in the diagram. However, for reasons of clarity, the exact definition of the calendar period units is not included. A date format, e.g. 21/6/1963, is just a shorthand notation of a calendar period.

According to the above, the information in a conceptual schema is time stamped using symbol period identifiers. No distinction is made between time periods and time points. The fact that the abstraction level of a SymbolPeriod stamp is e.g., *day* can be inferred by its constraining temporal relations. The fact that an entity is time-stamped only at the day abstraction level ill be represented by distinguishing between different SymbolPeriod subclasses according

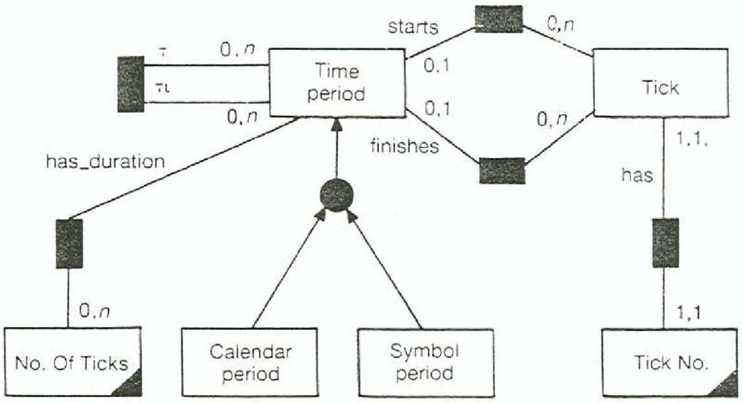


Figure 5. Time period metamodel.

to their abstraction level, i.e. SPD, SPM, etc. (this last notation is not represented in the example of Fig. 3). This form of constraint is a resolution constraint which, when applied to a SymbolPeriod class, restricts its members to calendar periods of the same duration.

It is suggested that it would be convenient, to represent directly in the conceptual schema, some other notions of time such as duration and periodic time. The consequence of this is that the expressive power of the ERT external formalism is increased and so is the readability of the schema. In (Theodoulidis *et al.*, 1990) there is a more detailed description of the time semantics of ERT.

In the ERT model, value classes and the IS_PART_OF relationships in a complex value class, should always be time invariant. This is because an aggregation or grouping of values is defined through the participating value components. These assumptions affect the way that an ERT scheme is mapped onto a relational schema. As discussed already, the validity period of a relationship should be a sub-period of the intersection of the existence periods of the involved entities. This does not hold for the ISA relationship as the existence period of the specialized entity should be a sub-period of the existence period of its generalization and that the ISA relationship is always time invariant.

Time stamping, when applied to derived ERT components, has slightly different semantics than usual. Since, the derived components are not stored by default, the interpretation of time stamps refers to their corresponding derivation formulas. That is, if a derived component is not time stamped then the derivation formula returns the value of it at all times, i.e. for every valid state of the database. Alternatively, for the time-stamped derived components, the derivation formula returns a value which is valid for the existence or validity period of this component, i.e. the derivation formula must refer to this period.

Finally, time stamping in a time varying IS_PART_OF relationship is translated to the following constraints. The dependency constraint in a time varying IS_PART_OF relationship means that:

1 the existence periods of the complex object and the component object should finish at the same time with the validity period of the IS_PART_OF relationship.

Also, the exclusiveness constraint is translated to the following.

2 If an object A is part of the complex objects B and C, then the period during which A is part of B should have an empty intersection with the period during which A is part of C.

Summarizing, the above time semantics permits us to keep historical information for the Universe of Discourse, include a strong vocabulary for expressing temporal requirements and also model the evolution of complex objects through time in a natural way.

The External Rule Language

In using a temporal logic rule-based language (TL) to control the processing of information, TEMPORA has allowed for the representation of business rules of an information system in a highly declarative manner. Business rules, and the corresponding aspect of the executable rules, may be classified into three different classes.

1 Action Rules, which imply some action must be taken if some condition holds, and will be modelled by a full TL rule of the form *condition→ action*.

2 Derivation Rules, which express that some facts hold if some group of other facts hold, and are used during execution to evaluate the condition part of TL rules, and are represented by a rule in the condition language of the TL of the form *condition derived←condition*.

3 Constraint Rules, which specify that some condition must not be violated. These are used during execution to validate the actions being taken. If the constraint would be violated, then the rule will *rollback*, and cause an error to be raised.

At the conceptual level there is not this distinction between different classes of rule, and so we introduce the notion of an External Rule Language (ERL), which can model different classes of executable rules in-uniform manner. We leave the decision of which class a rule belongs to until the design phase, and thus allow the specification of the rules in our system to closely mirror the business rules from which they are derived. This two level approach allows users of the system to inspect the rules in a form which they comprehend, but still gives the designer the procedural control over execution, in choosing the interpretation given to an ERL rule when translated into a TL rule. To summarize, the ERL rules closely match the business rules, and the TL rules closely match the procedural interpretation given to the business rule in the design phase.

The presence of the ERL also allows us two other important benefits.

1 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.

2 We may heavily sugar the syntax of the ERL to give a semi-natural language flavour, whilst leaving the internal rule language in a more concise form that programmers would desire.

ERL expressions

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

```
ERL_rule::=[ [WHEN<trigger_exp>] [IF<cond_exp>] THEN]<exp>
```

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

(a) `<exp>`
exp must always hold,

(b) `IF<cond_exp>THEN<exp>`
exp must hold whenever *cond_exp* holds,

(c) `WHEN<trigger_exp>THEN<exp>`
exp must hold when *trigger_exp* has just begun to hold,

(d) `WHEN<trigger_exp>IF<cond_exp>THEN<exp>`
exp must hold if *cond_exp* holds, and *trigger_exp* has just begun to hold.

Referencing the ERT Model

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.

```
ERL_data_access::=<entity/value name>[ (<variable>) ]
[[<relationship><ERL_data_access>{,<relationship><ERL_data
_access>}]]
```

As an example, consider the data access expression for the ERT of Fig. 6, which models an application that deals with the handling of arrangements about customers' accounts for a public utility organization.

The following expression finds all pairs of *account* references *a* and *instalment* references *i*.

```
account(a) [is_governed_by arrangement
        [comprises instalment(i)]]
```

Note that we need only give variables to the entities or values we are interested in finding information about, so for the above example we were able to omit a variable for the *arrangement* entity. Also note that the enclosure of *comprises instalment(i)* in brackets is necessary to indicate that we expect the relationship *comprises* to be between *arrangement* and *instalment*, and not between account and instalment.

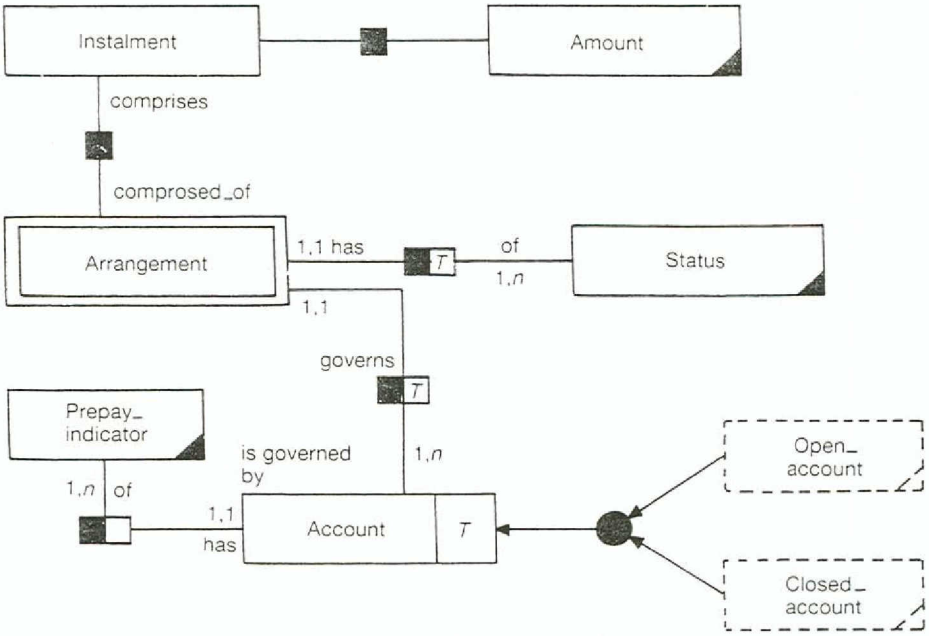


Figure 6. An example ERT model.

Collecting information

A set collection construct is provided, together with a group of set operators. This allows us to group all variable substitutions for which an expression holds, and perform operations such as COUNT to count the number of instances or INTERSECT to find common substitutions from two set expressions.

ERL_set_exp::={<*var*>{,<*var*>}|<*exp*>}

Thus we could find the number of accounts by the expression COUNT{x|account(*x*)}.

An example of the ERL and TL

As an example of the use of ERL in interpreting business rules and of the use of the TL in executing these rules, consider the application modelled in the ERT model of Fig. 6. In this example, an *arrangement* is an agreement between a customer and the public utility organization for the customer to make some set of fixed payments (*installments*) at given dates. Both the *instalments* and actual *payments* made are recorded in the database.

In the business rules listed here, the notion of a temporal database is used in order to be able to describe entities as being 'present' in the 'current' database or 'past' database, and thus phrase rules in terms of current records and old records. In fact, all information is present in one database, but the TL provides a mechanism for it to appear as a series of databases or 'states' over time (q.v. section on the TL).

(a) The number of accounts is limited to 1,300,000,

(b) At most 25,000 accounts shall have arrangements,

(c) An open account is one which exists in the current records,

(d) A closed account is one which does not exist in the records at present, but existed at sometime in the past,

(e) A clearby arrangement has only one instalment,

(f) A scheduled arrangement has more than one instalment,

(g) If it is 7 days after an instalment was due on a scheduled arrangement, and the sum of all payments made during the period of the arrangement is found to be less than the sum of all previous instalments and half of the last instalment, then the arrangement shall be terminated,

(h) If it is 7 days after an instalment was due on a clearby arrangement, and the sum of all payments made during the period of the arrangement is within £25 of the instalment amount, close the arrangement in the normal way.

The business rules (a)–(h) may be interpreted by an analyst into the following ERL rules, where all keywords of the language have been written in capitals. A short explanation of any new constricts of the language introduced, follows each example rule.

```
(a) COUNT{a|account(a)}<1,300,000
(b) COUNT{a|arrangement(a)}<25,000
(c) IF account(a) THEN open_account(a)
(d) IF NOT account(a) AND IN_PAST account(a)
    THEN closed_account(a)
      /* NOT<exp> holds for variable substitutions for which
      exp does not hold.
      IN_PAST<exp> holds for each instance of exp held in the
      past.*/
(e) IF COUNT {i| AT_ANY_TIME instalment(i)
    [comprises arrangement(a)]}=1
    THEN clearby(a)
      /* AT_ANY_TIME<exp>holds for each instance of exp holding at
      any time past, present or future. Note here that we derive
      an entity as holding in the present from information
      about entities which held in the past.*/
(f) IF COUNT {i|AT_ANY_TIME instalment(i)[comprises
    arrangement(a)]}>1
    THEN schedule(a)
(g) IF 7*DAYS AGO
      (instalment[value amount(install), belongs_to
        arrangement(a)]
      AND schedule(a)
      AND prev_instalments=SUM {old_v|
        IN_PAST instalment[value amount(old_v), belongs_to
          arrangement(a)]})
    AND payments=SUM{p|IN_PAST amount(p) [of account_movement
```

```
          [type payment, charged_to account[governed_by
            arrangement(a)]]]
        AND payments<prev_instalments+0.5*install
        THEN terminate_arrangement(a)
            /*<time>AGO<exp>holds for each instance of exp which
            held at time before the present. SUM<ERL_set_exp>finds
            the sum of all instances of the leading variable from the
            set expression variable list.*/
(h) IF 7*DAYS AGO instalment[value amount(instal), belongs_to
          arrangement(a)]
        AND clearby(a)
        AND instal-25<SUM{pay|IN_PAST amount(pay)[of
          account_movement[type payment, charged_to
          account[governed_by arrangement(a)]]]
        THEN close_arrangement(a)
```

To complete the simple example of the arrangement handling system, a possible interpretation TL is presented. It is during the design phase that the conceptual rules of the ERL are translated to the more procedural rules of the TL. For the purposes of this example it has been assumed that business rules (a) and (b) are constraint rules, (c)–(f) are derivation rules, and (g and (h) are action rules. The TL rules make reference to a Relational Database (RDB) schema and therefore a mapping is required from the ERT level to the RDB level (an automatic generator is being developed in the project). The mapping chosen in this example has stored a the values associated with each entity in a single table, with a surrogate field representing the entity, and a table for each relationship between two entities containing the surrogates for the entities.

```
(a) declare esb_account(_,_,_,_,_,_)=database
      (cardinality:0..1,300,000),
```

Constraints are not made to be part of a TL rule, but declared to the TL rule system, and used t control the updates made as part of the actions of a rule. As part of the design phase we ma also wish to include the constraint information as part of rule conditions, to prevent the possibility of an action attempting to violate the constraint. For instance, as part of the condition of rule which is intended to add a new account, we may check that the number of accounts alread present allows us to insert a new one without violating the constraint.

```
(b) declare arrangement(_,_)=database
      (cardinality:0..25,000),
(c) open_account(Acc)<esb_account(Acc,_,_,_,_,_),
```

A derivation rule can be stored as a clause of the PROLOG-like language used to evaluate T conditions.

```
(d) closed_account(Acc)⇐¬esb_account(Acc,_,_,_,_,_)△♦
      esb_account(Acc,_,_,_,_,_),
```

The ⧫ operator causes a search of past information to be made.

```
(e) clear_by (Arr)<count(•⧫instalments(Arr,_)▽◊instalment
    (Arr,_),1).
```

The *count* predicate finds as its second argument the number of instances for which the expression as its first argument holds. A TL formula of the form •⧫*x*▽ ◊*x* causes all past, present and future instances of *x* to be found.

```
(f) scheduled(Arr)⇐ count(•⧫instalment(Arr,_)▽◊instalment
    (Arr,_),n)△n>1.
(g) in_past(7*days,
                  (instalment(Arr,Last_Amount)
                  △arrangement(Arr,current)
                  △governed_by(Esb_Acc,Arr)
                  △•sum(Due,P instalment(Arr,Due),Total_Due)
                  )
             )
    △sum(Payment,P(movement(Esb_Acc,payment,
    Payment)△
    arrangement(Arr,current)),
       Total_Paid)
    △Last_Amount*0.5+Total_Due>Total_Paid
    →O terminate_arrangement(Arr).
```

A full TL rule models an action rule. The in_past predicate finds if the expression given as its second argument held at a time before the present indicated by the first argument. The O operator specifies that the actioon takes place immediately.

```
(h) in_past(7*Days,instalment(Arr,Amount))
    △clear_by(Arr)
    △governed_by(Esb_Acc,Arr)
    △sum(Payment,P(movement(Esb_Acc,payment,Payment)
    △arrangement(Arr,_)),Total_Paid)
    △Amount-25<Total_Paid
    →O close_arrangement(Arr)
```

CASE TOOL SUPPORT

The TEMPORA case environment is being implemented in a case-shell called RAMATIC, which is a 'meta-tool' for case tool implementation developed by the Swedish Institute for Systems Development (SISU). RAMATIC includes a number of features to facilitate the implementation of a case tool for a particular, graphics-oriented method. For typical specification methods, which are similar to SADT, ER-modeling, and hierarchical decomposition of data flows and processes, the time to create a case tool is in the order of 2–3 weeks.

RAMATIC can be used to capture, and store in an integrated fashion, a wide range of different types of specification, be they graphical, form-oriented, tabular or pure text. This is evidenced by the current use of RAMATIC in a number of real projects in industry, where different kinds of description and specification techniques are employed. It is also possible to include project control and quality control information such as design decisions, various annotations, information about designers/analysts, etc., in the specification. On the output side, various report forms can be defined as well as special checks to be performed. Various cross-reference matrices can be defined easily in order to display 'where used' and 'where created' information. For special analysis purposes it is simple to extend RAMATIC by analysis programs written in C or PROLOG. The coupling to these extensions may be more or less tight depending on whether RAMATIC invokes them or communicates with them via files.

In TEMPORA, a requirement of the CASE tool is to provide facilities for the diagnosis and analysis of the conceptual schema. To this end, the currently supported DBMS cs5 is supplemented by the PROBE system which is an Objected-oriented layer developed on top of BIM_PROLOG. Since the storage mechanisms of PROBE and RAMATIC are based on different approaches, a mapping from RAMATIC_cs5 internals to PROBE objects has been implemented. In a longer perspective RAMATIC has to be re-implemented to some extent in order to achieve a closer integration.

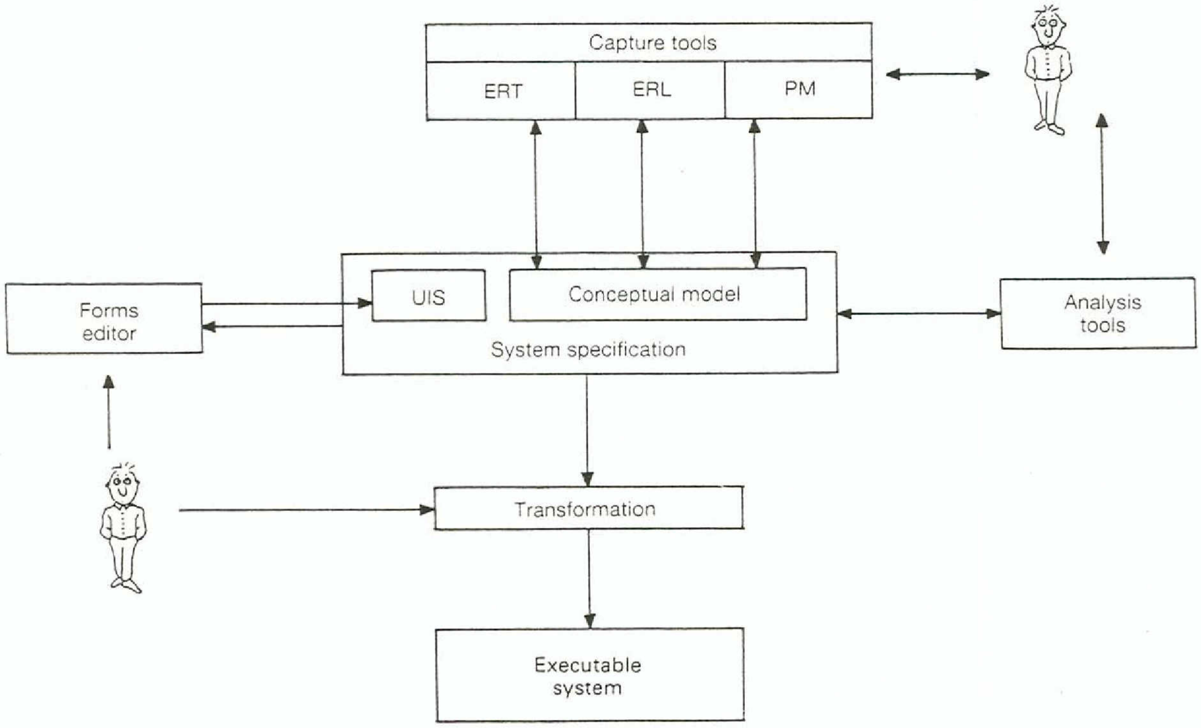


Figure 7. The TEMPORA case tool environment.

The architecture of the TEMPORA case tool environment is shown in Fig. 7. This is the environment by which one would develop applications according to the TEMPORA paradigm. The TEMPORA case tool includes a set of menus that drives the user through the different parts of the tool. However, there are separate components for each one of the capture and analysis tools, i.e. the ERT editor, the ERL editor and the Process Model editor, as seen in Fig. 7. Each of these components takes care of the capturing and analysis of a specific part of the application domain and it satisfies a set of requirements in terms of their functionality.

In addition to the capture and analysis components, the TEMPORA case tool also provides a forms editor. Currently, the exact format and functionality of the design tools and the way that these tools will be interfaced with the rest of TEMPORA case tool is being investigated.

RUN-TIME ENVIRONMENT

The non-temporal rule manager

As seen in Fig. 1, in the current TEMPORA architecture there are two sets of executable rules and also rule managers. The first one is the non-temporal approach that may be used for applications that do not require historical information in the database in contrast with the second one, which is called the temporal approach.

For an application to run in non-temporal mode, certain restrictions need to be adhered to during the analysis and design phases. These restrictions relate to the ERT model and the ERL and TL rules. The restriction to ERT is that the time stamps are not considered (thus reducing the model to an extended ER model), whereas the restriction to ERL and TL rules is that the only temporal references concern the next time slot, which is equivalent to performing modifications only on the current database, and not on some future/past database. Furthermore, the action rules should always mention the trigger (WHEN part) to indicate the specific circumstance under which the rule should be considered. Given these restrictions at the specification level the application can benefit at run-time from a non-temporal rule manager. Figure 8 gives a block diagram from the run-time subsystem showing also the main flow of information between modules. The main characteristic of this approach is that the underlying relational DBMS does not necessarily possess a built-in rule management facility. Consequently, any conventional RDBMS could be used together with our extension module.

All rule management activity is undertaken by the extension module (EM) while database management is accomplished through the RDBMS. The EM consists of four main sub-modules: the top level driver (TLD), the interface mechanism (IM), the external rule manager (ERM) and the rollback mechanism (RM).

The top level driver module (TLD) coordinates the behaviour of all the other modules between each-other as well as in connection to the RDBMS, while at the same time it handles the interaction between the user's environment and the run-time subsystem. The interface mechanism (IM) handles deductive query processing. According to our choice of an interpeted or a compiled approach it could incorporate either a coupling or a compilation mechanism. In all cases IM interfaces RDBMS to TLD and subsequently to ERM. The external rule manager (ERM)

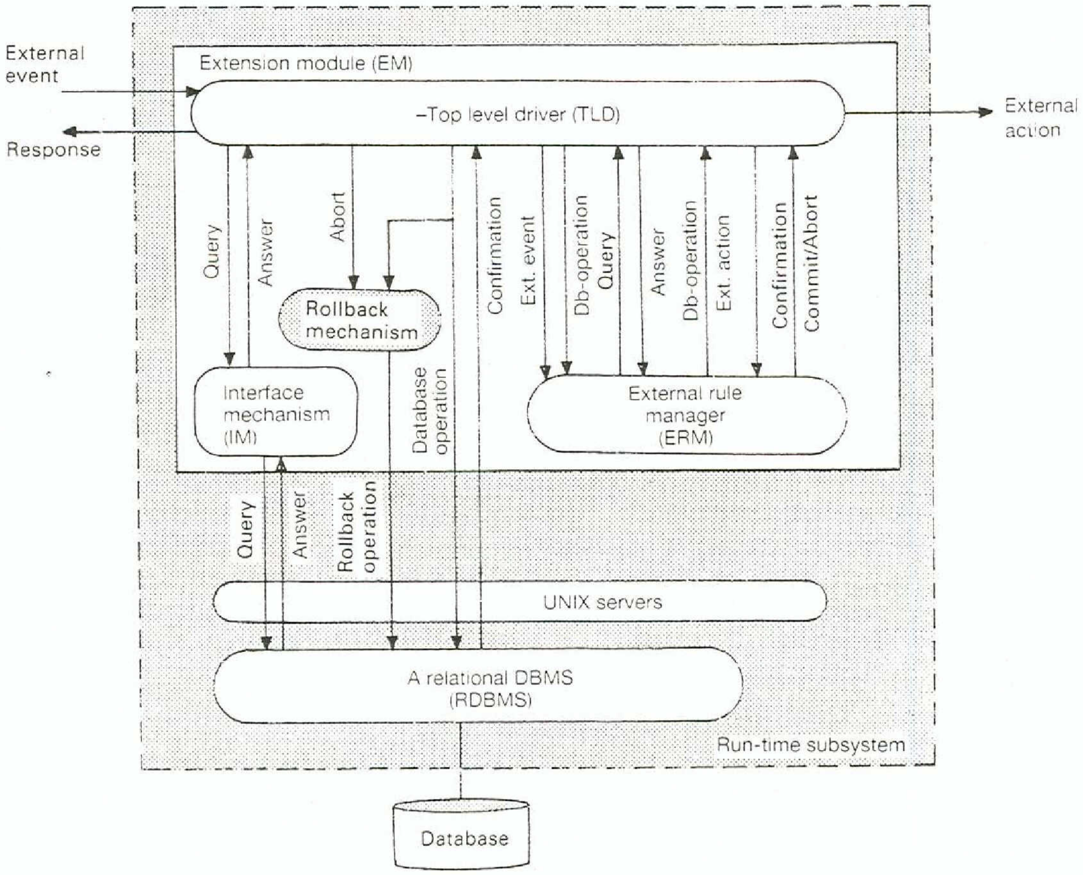


Figure 8. The run-time system architecture.

handles the entire rule management and processing functionality of our system. It uses TLD as a communication bus for interaction with the rest of the run-time subsystem as well as the environment of the user. The rollback mechanism (RM) finally restores a database-consistent state after a transaction aborts (as a simpler alternative, we could consider the rollback mechanism as part of the standard features offered by the RDBMS).

The communication between the EM and the RDBMS is done through UNIX servers, and within each module according to its local protocol. Transactions are initiated by external events detected by the TLD inside EM.

When TLD detects an external event (database operation, external signal) submitted by users, application programmes or the system clock, it converts this event to atomic elements. Parts of the external event that happens to be database operations are logged inside the rollback mechanism and then applied to the basabase. The RDBMS then confirms an operation's validity (e.g. in connection to the database schema definition). Information about the event is then passed to ERM which then detects and activates the relevant database level rules.

Database level rule processing is undertaken by ERM which, during database level rule condition evaluation, sends queries to RDBMS via TLD and the IM. Answers are collected and sent back from the RDBMS to IM, to TLD and then back to ERM.

Firing of dynamic rules involves external actions and/or database operations to be applied. External actions are then sent to the environment of the user via TLD while database operations are logged inside RM and then sent to RDBMS via TLD. The RDBMS again confirms an operation's validity.

Transactions are initiated by external events but subsequent triggering cycles are initiated by database operations caused by database level rule action part application. TLD informs ERM about all these internally generated operations. Triggering cycles continue until transaction termination is either successful or unsuccessful. Successful transaction termination occurs when no more active database level rules remain inside ERM. This situation is detected by TLD when receiving a commit signal by ERM. Successful transaction termination is followed by commitment to the current database state (all database operations related to the current transaction are made permanent).

Unsuccessful transaction termination occurs either when an explicit rollback operation is detected inside ERM and an 'abort' signal is sent from ERM to TLD or an 'invalid operation' confirmation signal is sent by RDBMS to TLD. In both these cases, TLD activates the rollback mechanism (TLD transmits an 'abort' signal) which 'undoes' all database operations related to the current transaction (when using a RM which is built-in to the RDBMS, then TLD sends an 'abort' signal directly to the RDBMS). Transaction termination (either successful or unsuccessful) is followed by a response to the external event that intitiated the transaction.

The rules supported by the non-temporal rule manager are of the general form:

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

In this 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 with 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`, described above. In Fig. 9, a simple rule that 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) is shown.

```
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 where the event specifies a modification (insert/delete/update) of a single entity in the ER diagram, and the conditions specify relationships between entities which must hold for the actions (further modifications and/or calls to foreign procedures) to take place.

The non-temporal rule manager is responsible for the run-time execution of the QL−>AL rules. According to this 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 program, system clock).
2 *Operation blocks* submitted by an external agent (user or application program).

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).

This is done, however, only in the case of operations submitted by external agents (users, application programs, etc.). Operations found in the action part of rules are treated differently. We apply action parts of rules as atomic units using all rules as if they were in mode-d (similar to the system in (Widom, 1989)).

The above functionality scheme is justified by the following rationale. User input is frequently not well thought out and needs supervision. The adopted system provides this supervision through the use of mode-i and mode-d rules. Additionally this supervision can be made tighter or looser. Tighter supervision can be obtained by putting more rules in mode-i, while looser supervision is obtained by putting more rules in mode-d. Rules on the other hand, could be considered as being internal agents who submit their action part for application. For operations inside the action part of rules we do not provide both mode-i and mode-d rule consideration facilities. Instead, all rules are considered as if they were in mode-d. This is because the action part of rules is written by systems analysts and it is assumed that it is 'well thought' and 'tested' and consequently, it does not need the constant supervision that mode-i offers. Avoiding mixed mode semantics for the action parts of rules greatly simplifies the model without a significant loss of function.

The Temporal Rule Manager

In TEMPORA action rules can refer to the temporal dimension of the database and thus are composed of two temporal components: the TQL (Temporal Query Language) and the TAL (Temporal Action Language). A TQL query is evaluated on the database and is concerned with the condition part of the rule. The action part is a TAL formula specifying actions to be taken.

The function of the Temporal Rule Manager (TRM) is to execute these rules in such a way that the system matches its specification. In other words, the TRM evaluates the queries and performs the actions necessary to ensure that the system actually behaves as stated in the rules.

As a first approach, it is assumed that the system is powerful enough to process an arbitrary large set of rules within a tick. This assumption will allow us to define a theoretical execution mechanism independently of any possible implementation constraint. Of course, in a real situation this assumption cannot be satisfied. Therefore, the basic model needs to be refined to make it practically feasible.

Assuming that the previous hypothesis is satisfied, the basic execution mechanism can be viewed as the infinite repetition of an elementary cycle starting at each tick. The work peroformed during each cycle can be described as follows.

1 The Query Module (QM) evaluates the condition part of each rule.
2 For each rule R_i, the QM generates the corresponding action set* A^S_i.
3 The QM sends the global action set ($A^S_1 \cup A^S_2 \cup \ldots \cup A^S_n$) to the Action Module (AM) and stops.
4 The AM analyses the different TAL formulae of the global action set and determines the sequence of actions to be performed directly. This operation includes several steps: consistency checking, scheduling, etc. The AM also keeps all (partial) TAL formulae describing actions to be taken at future ticks.
5 The AM sends the action sequence to the Transaction Module (TM) which performs the actions immediately.
6 Go to step 1 and start a new cycle at the next tick.

This simple mechanism is sufficient to ensure that the system will match the specification described by the rules, because:

(a) The system re-evaluates all the rules at each tick and performs the necessary actions,
(b) The tick is the smallest time unit representable (i.e. everything is fixed in the system within a tick), and thus our execution mechanism will see all changes of the database.

If the choice of the tick as the basic cycle length appears to be a sufficient condition to ensure that the system will match its specification, it will usually not be necessary to re-evaluate all the rules at every tick. This particular aspect is currently under investigation.

The architecture of the TRM can be represented by Fig. 10. Each sub-module has an auxiliary database attached to it, which contains all information necessary for the module to work properly.

The refined execution mechanism

Temporal Rule Manager

From a practical point of view the basic model has many limitations not least the fact that the

*The action set A^S_i for a query Q_i is the set of TAL formulae $\{A^1_i ... A^k_i\}$ obtained from A_i for all instatiations of variables for which the query part is true. If the query cannot be satisfied, the action set is empty.

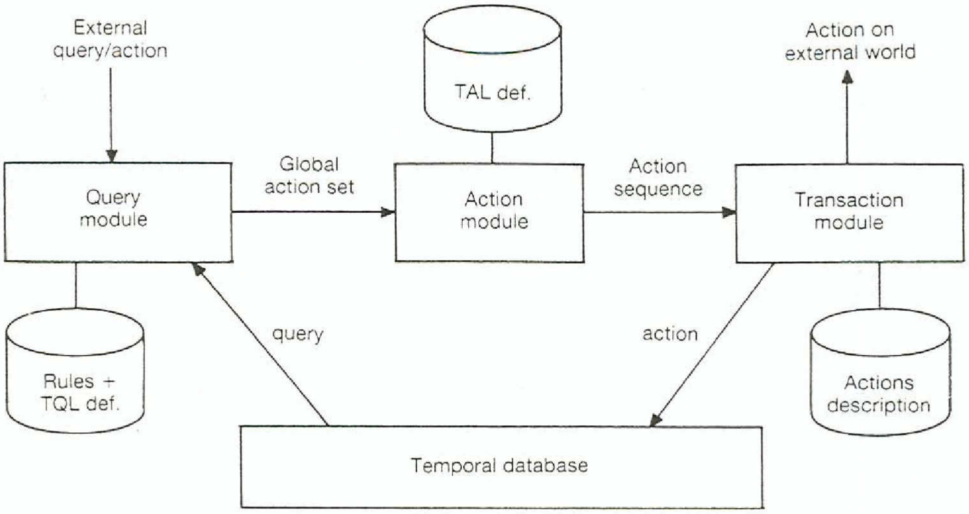


Figure 10. Temporal rule manager architecture.

number of rules that can be processed within the duration of one tick (whatever duration we choose for the tick) is finite. Therefore, the execution mechanism can be modified to take into account a possible delay in the temporal modules with respect to real time (i.e. the time will not be finished at the end of the tick). The existence of a delay is especially important here, because we deal with temporal rules. For instance, if we consider the evaluation of a query, the evaluation begins in the current state of the database, which has been defined as the state that contains the current real time. It is obvious that starting the evaluation in a different state can lead to a different answer. Therefore, a query should always be evaluated at the right real-time.

The only reasonable way to solve this problem is to introduce two time references:

(a) the actual real time, denoted t_R, and

(b) a temporal module time, denoted t_{TM}, which is considered as the real time by the different temporal modules, but can actually be delayed with respect to t_R.

More precisely, t_{TM} can be defined as follows: after each evaluation cycle, t_{TM} is incremented by one tick; if t_{TM} is in advance of t_R, then the system waits until $t_R = t_{TM}$ to begin the next cycle, otherwise the next cycle begins immediately.

When the system is (almost) idle, everything happens as if t_R was equal to t_{TM}. If the load of the system increases, the virtual clock t_{TM} slows down and t_{TM} is delayed with respect to t_R. However, if the system is not undersized, we can reasonably expect the system to catch up when the load decreases.

The use of the virtual time reference t_{TM} for the temporal modules allows us to use our basic execution mechanism on an actual computer system without any modification, except for:

$t_{TM} = t_{TM} + 1$ tick
Wait while $t_{TM} > t_R$ then go to step 1.

The assumption under which the system can process an arbitrary large number of rules within a tick is verified because a tick no longer has a fixed length. Its length may vary according to the amount of work to be done.

The only impact of this modification on the behaviour of the system is that when $t_{TM} < t_R$, the system does no longer behave in 'real time', i.e. at the real time t_R, the model present in the database is not up to date but corresponds to the model at time t_{TM}. Therefore, in most cases, we will have to buffer external requests until t_{TM} is greater than the time of submission, in order to make sure that these requests are evaluated on the correct model.

Taking the architecture of Fig. 10, we only need to add one input queue for every possible external input source (external query, action, signal, etc.). For example, when a query is submitted to the system, it is placed into the query queue and all the references to 'NOW' are instantiated with the time (t_R) of submission. The queries of the queue are processed by the system (i.e. submitted to the QM) when t_{TM} is greater than the time of submission.

CONCLUSIONS

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 to effect changes at the strategic level of organizations.

The process of software development can be viewed as a sequence of model-building activities. The quality of each set of models depends largely on the ability of a developer to extract and understand knowledge about an application domain which needs to be acquired from a diverse user population (Loucopoulos & Champion, 1988). However, current technology does not provide any powerful formalisms or tools to support such a view. The current fragmentation in development approaches has resulted in a situation where methodological knowledge is difficult to obtain and use in a constructive manner.

It is also becoming obvious that current conceptual modelling formalisms are oriented towards the functional specification of the software system rather than the definition of the problem domain (Greenspan, 1984). If improvements are to be made in the quality of software then the knowledge about the application domain must be formalized and explicitly encoded. To this end TEMPORA follows the premise that information system development is about formalizing 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.

The TEMPORA project is attempting to provide a better approach to building systems through the development of a software process and supporting tools which will explicitly accommodate those parts of a system that correspond to those elements of organizational policy which in essence impose changes to the structure and operation of information system. Traditionally, ANSI/SPARC-style data management architectures and recent developments in HCI management tools have enabled the separation of system oriented elements from procedural code. The main initiative of the TEMPORA paradigm has been to extend this approach to include organisational policy. In particular, TEMPORA seeks to separate out and explicitly maintain throughout the software life-cycle, the notion of policy, as described by *constraint, deviation* 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 organization and their structural relationships (using the ERT model) and the rules that are expressed by references to these objects (using the ERL). 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.

ACKNOWLEDGEMENTS

The work reported in this paper 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 Telecom.

This paper summarizes the contribution to the TEMPORA project of the following: R. Anderson, P. Bergsten, J. Bubenko Jnr, G. Diakonikolaou, R. Jenssen, M. Niezette, R. Owens, D. Pantazis, U. Persson, A. H. Seltveit, G. Sindre, A. Sølvberg, U. Sundin, G. Tzialas, P. Vasey, R. Venken, R. Wohed and P. Wolper.

REFERENCES

Allen J.F. (1983) Maintaining knowledge about temporal intervals. *CACM*, **26**(11) Nov.

Anderson, M. & van Assche, F. (1986) Report on task AI: research into the ability to use rules to describe the business and its activities. *Internal Report EQ28/R2/Final*, James Martin Associates, Brussels, Belgium.

Batini, C. & Di Battiste, G. (1988) A methodology for conceptual documentation and maintenance. *Information Systems*, **13**(3), 297–318.

Greenspan, S.J. (1984) Requirements modeling: a knowledge representation approach to software requirements definition. *Technical Report No. CSRG-155*, University of Toronto.

Kim, W., Banarjee, J., Chou, H.T., Garza, J.F. & Woelk, D. Composite object support in object-oriented database systems. In: *Proceedings of the 2nd International Conference on Object-Oriented Programming Systems, Language and Applications*, Orlando, FL.

Kim, W., Bertino, E. & Garza, J.F. (1989) Composite objects revisited. *Sigmod Record* **18**(2), 66–79.

Ladkin, P. (1987) Logical time pieces. *AI Expert* Aug, 58–67.

Lorie, R., Plouffe, W. (1983) Complex objects and their use in design transactions. In: *Proceedings of Databases for Engineering Applications, Database Week 1983* (ACM), San Jose, CA.

Loucopoulos, P. (1989) The RUBRIC Project-Integrating E-R, Object and Rule-based Paradigms, Workshop session on Design Paradigms. *European Conference on Object Oriented Programming (ECOOP)*, 10–13 July, Nottingham, UK.

Loucopoulos, P. and Champion, R. (1988) A knowledge based approach to requirements engineering using method and domain knowledge. *Journal of Knowledge-Based Systems*, June. **1**(3), 179–187.

Maddison, R. (1983) *Information System Methodologies*, Wiley-Heyden.

Olle, T.W. *et al.* (eds) (1983) *CRIS-Information System Design Methodologies: A Comparative Review*, North-Holland Publishing Co., Amsterdam.

Olle, T.W. *et al.* (eds) (1986) *CRIS3-Improving the Practice*, North-Holland Publishing Co., Amsterdam.

Rabitti, F., Woelk, D. & Kin, W. (1988) A model of authorization for object-oriented and semantic databases, In: Proceedings of the International Conference on Extending Database Technology, Venice, Italy, March.

Theodoulidis, C., Wangler, B. & Loucopoulos, P. (1990) Requirements specification in TEMPORA, In: *Proceedings of the 2nd Nordic Conference on Advanced Information Systems Engineering (CAiSE90)*, Kista, Sweden.

van Assche, F., Layzell, P.J., Loucopoulos, P. & Speltincx, G. (1988) Information systems development: a rule-based approach. *Journal of Knowledge Based Systems*, September, **1**(4), 227–234.

Villain, M.B. (1988) A system for reasoning about time. *Proceedings of AAAI-82*, Pittsburgh, USA.

Villain, M.B. & Kautz, H. (1986) Constraint propagation algorithms for temporal reasoning. *Proceedings of AAAI-86*.

Widom, J. & Finkelstein, S.J. (1989) A syntax and semantics for set-oriented production rules in rela tional database systems, *SIGMOD Record*, Vol. **18,** No. 3.

Biographies

Pericles Loucopoulos is Professor of Information Systems at the Department of Computation, UMIST. His research interests include development methods for data intensive systems, requirements engineering and databases. His research work has been supported by grants from SERC/Alvey, and the Commission of the European Communities under the ESPRIT and AIM programmes. He is a Fellow of the British Computer Society and a member of IEEE. He is the author and co-author of three books and over 50 papers.

Peter McBrien holds a BA in Computer Science from St Johns, Cambridge. He worked at Racal on realtime radar simulators and at ICL on the Alvey 'Pure Logic Language' project. He is currently a Research Assistant at Imperial College. His research interests include temporal databases, graph re-writing for logic languages, executable temporal language and meta-level programming.

Francois Schumacker is an Inginuer Civil Electrician (Informatique) and holds the Degree of Electrical Engineer in Computer Science from the University of Liege. He currently works as a Research Engineer in the Computer Science Department of the University of Liege. His research interests are in the area of temporal logics and temporal databases.

Babis Theodoulidis holds a BSc in Computer Science from the Univeristy of Patras, a MSc from the University of Glasgow and a PhD from UMIST. He currently works as a Research Assistant in the Department of Computation, UMIST. His research interests are in the areas of design approaches to data-intensive systems, conceptual modelling and temporal databases.

Vassili Kopanas holds a BSc in Electrical Engineering from the National Technical University of Athens, and a MSc from UMIST. He is currently Research Assistant in the Department of Computation, UMIST. His research interests include conceptual modelling, deductive databases and active databases.

Benkt Wangler is a technical manager at SISU and a faculty member of the University of Stockholm. He has been involved in many R&D projects, in collaboration with industry, in the areas of systems development and CASE. His research interests are in the areas of databases, CASE and conceptual modelling.