08:24:30 OCA PAD INITIATION - PROJECT HEADER INFORMATION 06/26/92

Active

Project #: C-36-687 Cost share #: Rev #: 0
Center # : 10/24-6-R7526-0A0 Center shr #: OCA file #:
Work type : RES
Contract#: F30602-92-C-0092 Mod #: Document : CONT
Prime #: Contract entity: GTRC

Subprojects ? : N CFDA:
Main project #: PE #:

Project unit: COMPUTING Unit code: 02.010.300
Project director(s):
NAVATHE S B COMPUTING (404)894-3152

Sponsor/division names: AIR FORCE / GRIFFISS AFB, NY
Sponsor/division codes: 104 / 023

Award period: 920612 to 930611 (performance) 930711 (reports)

Sponsor amount	New this change	Total to date
Contract value	22,000.00	22,000.00
Funded	22,000.00	22,000.00
Cost sharing amount		0.00

Does subcontracting plan apply ?: N

Title: MODELING OF DATABASE CONSTRAINTS IN ACTIVE DATABASES

PROJECT ADMINISTRATION DATA

OCA contact: Brian J. Lindberg 894-4820

Sponsor technical contact

DR. RAY LUIZZI
(315)000-0000

ROME LABORATORY
C3CA, BUILDING 3
GRIFFISS AFB, NY 13441-5700

Sponsor issuing office

MS. JANIS NORELLI
(315)330-2326

ROME LABORATORY
DIRECTORATE OF CONTRACTING/PKRD
GRIFFISS AFB, NY 13441-5700

Security class (U,C,S,TS) : U ONR resident rep. is ACO (Y/N): Y
Defense priority rating : DO-A7 GOV'T supplemental sheet
Equipment title vests with: Sponsor X GIT
HOWEVER, NONE PROPOSED OR ANTICIPATED.
Administrative comments -
INITIATION OF PROJECT C-36-687.

GEORGIA INSTITUTE OF TECHNOLOGY
OFFICE OF CONTRACT ADMINISTRATION

NOTICE OF PROJECT CLOSEOUT

Closeout Notice Date 07/20/93

Project No. C-36-687 Center No. 10/24-6-R7526-0A0

Project Director NAVATHE S B School/Lab COMPUTING

Sponsor AIR FORCE/GRIFFISS AFB, NY

Contract/Grant No. F30602-92-C-0092 Contract Entity GTRC

Prime Contract No. _____

Title MODELING OF DATABASE CONSTRAINTS IN ACTIVE DATABASES

Effective Completion Date 930611 (Performance) 930711 (Reports)

Closeout Actions Required:	Y/N	Date Submitted
Final Invoice or Copy of Final Invoice	Y	
Final Report of Inventions and/or Subcontracts	Y	
Government Property Inventory & Related Certificate	Y	
Classified Material Certificate	N	
Release and Assignment	Y	
Other _____	N	

CommentsEFFECTIVE DATE 6-12-92. CONTRACT VALUE $22,000.

Subproject Under Main Project No. _____

Continues Project No. _____

Distribution Required:

Project Director	Y
Administrative Network Representative	Y
GTRI Accounting/Grants and Contracts	Y
Procurement/Supply Services	Y
Research Property Managment	Y
Research Security Services	N
Reports Coordinator (OCA)	Y
GTRC	Y
Project File	Y
Other CARL BAXTER-FMD	Y
FRED CAIN-OOD	Y

NOTE: Final Patent Questionnaire sent to PDPI.

C36-687

FINAL REPORT

Modelling of Database Constraints in Active Databases

S.B. Navathe A.K. Tanaka

Georgia Institute of Technology
College of Computing
sham@cc.gatech.edu
(404) 853 0537

REPORT SUBMITTED TO
ROME LABORATORY
GRIFFISS AFB, NY 13441-5700

May 1993

Contents

Section 1

INTRODUCTION

This report deals with the problem of specification, modeling, and enforcement of constraints in databases. The active database area is emerging as a viable alternative for implementation of large scale database applications, particularly those involving data that needs close monitoring and control due to its dynamic nature. Applications in monitoring of personnel, equipment, materials etc. need capabilities available in modern database systems. The entire area of command and control applications is likely to benefit immensely from the emerging "active database" technology.

The term "active" has two connotations: first, in contrast to "passive," it implies that the database system has a component that allows it to actively perform changes within the database , and possibly to the environment consisting of other data, users, and equipment. The second connotation can be traced to the database actively offering information to the user whenever information of interest "happens," as opposed to being only "reactive" to a user's request whenever the user presents one. It is also possible to treat the "active" nature of a database as being equivalent to "dynamic" or changing constantly. This contrasts with the typical "static" nature of a database where data tends to remain constant unless changed explicitly by an outside intervention.

The currently available capabilities in database management systems are limited , but are likely to be expanded in the future very rapidly. One facility is known as triggers which are activated upon the occurrence of certain events, and which automatically cause actions within the database. For example, systems like SYBASE or INTERBASE allow triggers to be defined and have a similar style of trigger implementation. But the facility is limited by the number of triggers that can be defined to go with a relation, or the level of nesting possible. The so-called knowledge-management extension of INGRES allows rules to be defined and procedures to be invoked as a result of the firing of rules. The procedures in turn may give rise to new rule firings. The net result of these rules and triggers shows up in terms of changing some data values, or sending some control signals to other hardware for process control type of applications, or sending alerting messages to human decision makers. Both the trigger and rule facilities are quite powerful and give rise to the so-called "active" nature of the database system.

Section 2 of this report presents an enhanced conceptual data model which is based on

the popular entity relationship (ER) data model. We have enhanced it with a capability of modeling active database behavior at a higher level than the event-condition-action rules used in the Sentinel system [Cha91]. We have also proposed a diagrammatic convention to go with this model which may need further work in terms of implementing a diagramming and conceptual schema editing tool based on it. We present a concise, high level specification language and then show how it can be mapped into the facilities of an active commercial DBMS such as INGRES.

Database constraints are inherently declarative and can be classified in different ways. In section 3 of this report we address three different types of constraints: integrity constraints, the invariant properties of the model as constraints, and dynamic constraints that deal with a change of state that occurs during update operations. Our contention is that maintenance of constraints can be accomplished by deriving appropriate "active database behavior" in the conceptual schema that are translated into executable rules or triggers. These rules or triggers cause the appropriate "repairing" process that takes care of "fixing" the database so that it is consistent with the constraints.

The facility we have described here can be thought of as a part of the design tools that are needed to exploit the active capabilities in a database management system. The specification of active behavior in the form of events and rules in the $(ER)^2$ model is a design time activity. With a mapping tool , these are mapped into the actual functionality of the proposed DBMS. In future, this whole activity may be considered to be a function of the mediator which is responsible for enforcing constraints that the user specifies, and translating them to system executable rules or triggers. The $(ER)^2$ model may be considered as the user's view or "window" on the underlying active database. The user can continue to manipulate this view with the help of a possible future tool for $(ER)^2$ schema manipulation and editing. The mediator will be responsible for "reflecting" these changes in the actual DBMS.

In section 4 of this report we have placed this work in proper context and have pointed out a large number of possible directions for extending this work.

Section 2

$(ER)^2$ MODEL

In this section, we propose an extension to database conceptual modeling using the Entity-Relationship (ER) approach to incorporate active database behavior in the form of events and rules.

The ideas presented here have first been delineated in [TNCK90, TNCK91] and further developed in [Tan92]. As the basis for the proposed extension, we adopt the variant of the ER model used in the Lawrence Berkeley Laboratory (LBL) tools [SM91, MF91] that includes generalization/specialization and full aggregation as relationships involving relationships, thus requiring directed arcs in the ER diagram to denote inter-object connections. The choice of a particular variant of the ER model does not disturb the incorporation of active database behavior in the conceptual schema, because the new dimension is orthogonal to the data abstractions of the model.

First, we define the concepts of events and rules, and present a syntax for active behavior specification at the conceptual level in terms of events and rules in section 2.1. Then we describe the operational semantics of the language in section 2.2. We introduce $(ER)^2$ diagrams (ER diagrams with events and rules) in section 2.3. In section 2.4 we show how the active behavior specified in the $(ER)^2$ model can be algorithmically mapped into language constructs at the relational DBMS level.

2.1 Events and Rules as ER objects

In the ER approach, the basic objects are entities and relationships that, along with their attributes, model the objects of the real world and their properties. Figure 2.1 shows how these concepts are viewed in a meta-schema, i.e., a meta-ER-diagram of the ER model itself. In the figure, "ENTITY" and "RELATIONSHIP" are specializations of a generic meta-entity "ER_OBJECT", to which they are connected by "Is_A" arcs; the meta-entity "ATTRIBUTE" is identification dependent on "ER_OBJECT", so it is connected to "ER_OBJECT" by an "ID" arc. The meta-relationship "ER_Connection" means the different types of directed arcs that may occur between ER objects. Possible connections are:

- Inter-entity connections: "Is_A", and "ID" arcs.

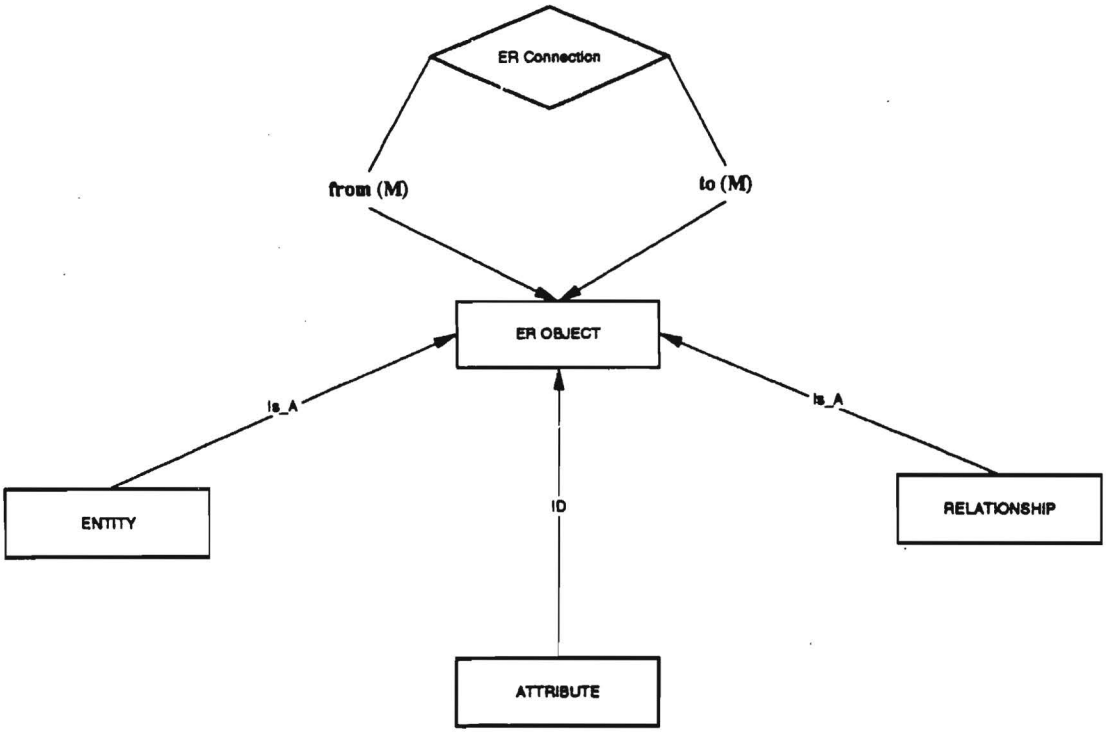

Figure 2.1: Meta-ER-diagram of the ER Model

- Connections between entities and relationships and inter-relationship connections: association arcs showing cardinality, participation constraints, and roles of the participating objects (e.g., in figure 2.1, "from" and "to" are roles of "ER_OBJECT" in "ER_Connection", while "M" means the cardinality "many" of "ER_OBJECT" in "ER_Connection").

In our approach, we view the real world as constituted by entities, relationships, events and rules, all primitive objects of the model. While entities and relationships, along with their attributes, represent the structural aspects of the information system being modeled, events and rules represent the active behavior that controls the states of the data objects and their attributes. We call the resulting model as the Entity-Relationship model with Events and Rules, or $(ER)^2$ model for short. The abstract construct that extends the ER model has the following grammatical form[1]:

[1] We use a BNF-like notation for syntax, where non-terminals are denoted in italic lower case letters, while words in non-italic lower case and upper case letters denote terminals. Single-quoted characters such as ':' are terminal delimiters whereas the rest are meta-characters. Square brackets [...] are used to denote optional constructs, and the notation [...]* denotes zero or more repetitions of the enclosed construct. '|' is used to delimit alternatives and curly braces { ... } denote one from a number of enclosed alternatives.

Page intentionally left blank

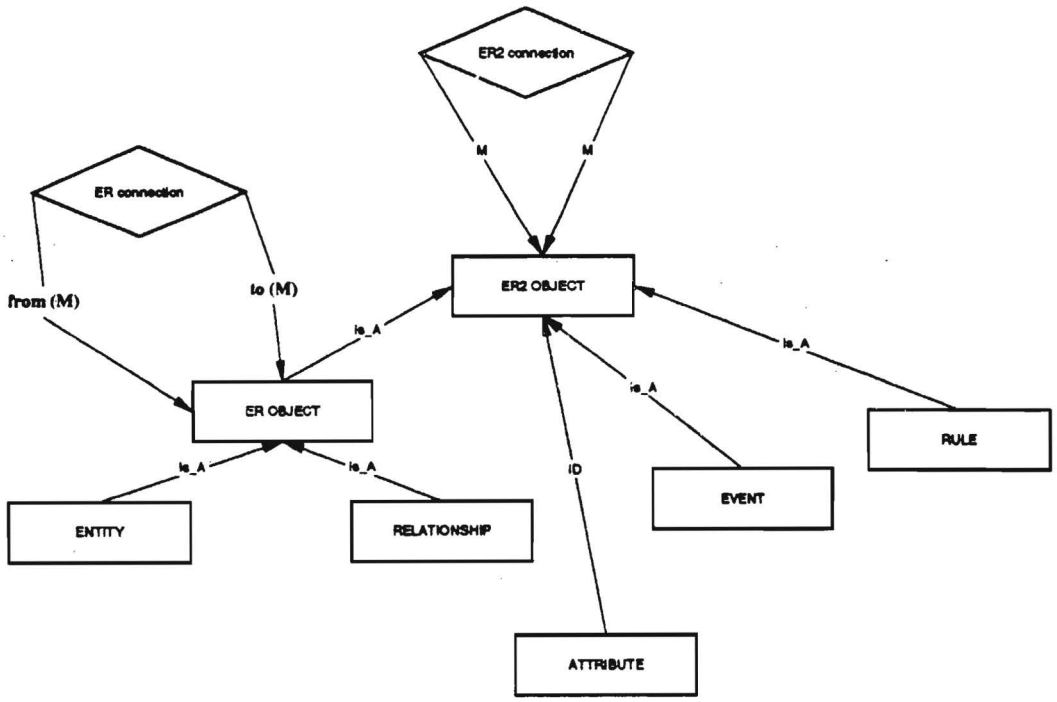

Figure 2.2: Meta-ER-diagram of the $(ER)^2$ Model

rules are always bound dynamically (at transaction execution time). For example, the event "all employees working on project Alpha have been deleted" is an instance of the event class "EMPLOYEE deleted"; they are bound at the time of the occurrence of the event. Note also that an event instance does not correspond necessarily to a single object instance; in general, it refers to a set of object instances (in the example, the set of employees working on project Alpha). The concept of rule classes/instances relates to that of event classes/instances. In the same example, a rule that is fired by the event instance "all employees working on project ALPHA have been deleted" is an instance of a rule class whose firing event class is "EMPLOYEE deleted"; this rule class may have different instances for different sets of deleted employees (i.e., event instances). Classes of events and rules are specified statically, along with the $(ER)^2$ schema; as we pointed out before, instances are determined dynamically, at transaction execution time, i.e., at event occurrence time. As with entity and relationship, sometimes we will be using the terms event and rule ambiguously, either referring to (event and rule) classes or to (event and rule) instances. Because every event instance is a unique occurrence and so is its associated rule instance, there is no ambiguity. The context determines whether we are referring to classes or instances.

2.1.1 Events

We use the concept of event as "the actual outcome or final result" [American Heritage Dictionary, Second College Edition, page 470]. An event is something that happens at a point in time, and, theoretically, has no duration [RBP+91]. In fact, instead of unifying the notion of action/event as commonly found in the literature, we distinguish between the occurrence of an event and the action that caused it (i.e., events occur when the associated actions have been executed by some agent). For example, the action "update balance" causes the occurrence of the event "balance updated". An action is, in general, denoted by a verb in imperative form; its execution takes time. However, an event, in general, is denoted by a verb in past participle tense, and its occurrence is just a point in time. This distinction is fundamental for the approach we propose.

Events may logically precede or follow one another or may be unrelated. There are two types of ordering of events to be considered: a causal ordering and a temporal ordering. The first one relates events of different types (e.g. the event "flight X landed" cannot occur before the event "flight X taken off"). Not all pairs of events bear this relationship. Causally unrelated events are said to be concurrent and can occur in any order because they have no effect on each other. The temporal ordering, on the other hand, is based on the linear ordering of the time of occurrence of the events.

The time of occurrence is an inherent attribute of every event. Every event has a unique time of occurrence or time stamp associated with it, which is assigned at the commit time of the action that causes it. A time stamp is in fact a unique identifier of an event; it implies a canonical precedence order among events. It represents the registration or assertion time of an event, also called transaction time in temporal modeling literature [SA85] . The granularity with which the time of occurrence of events can be represented is application and implementation dependent. Usually the time stamp is not implemented as a real clock value but as a unique serial integer for reasons of simplicity. However, this integer value can easily be mapped into a real clock value such that a level of indirection is created. As far as the activation of the behavior is concerned, we consider that the time of detection of an event is the same as its time of occurrence. This may not be true for actual implemented systems, but it is not a problem in conceptual modeling as long as the order is preserved.

Some events simply signal that something has occurred (e.g., "machine out of money", "engine stopped"), while others carry information in the form of event attributes, similar to the attributes of data objects. For example, in the event "salaries of employees working on project Alpha have been updated", the affected employees' names and salaries (and other attributes of employees) are conveyed through the fired rule (or rules) as attribute values of the event.

We distinguish events that occur on data objects or attributes stored in the database (database events) from those events that are external to the database, usually generated by application programs (external events). Figure 2.3 shows a third classification of events (system events), which are signals generated by the underlying system such as interrupts and

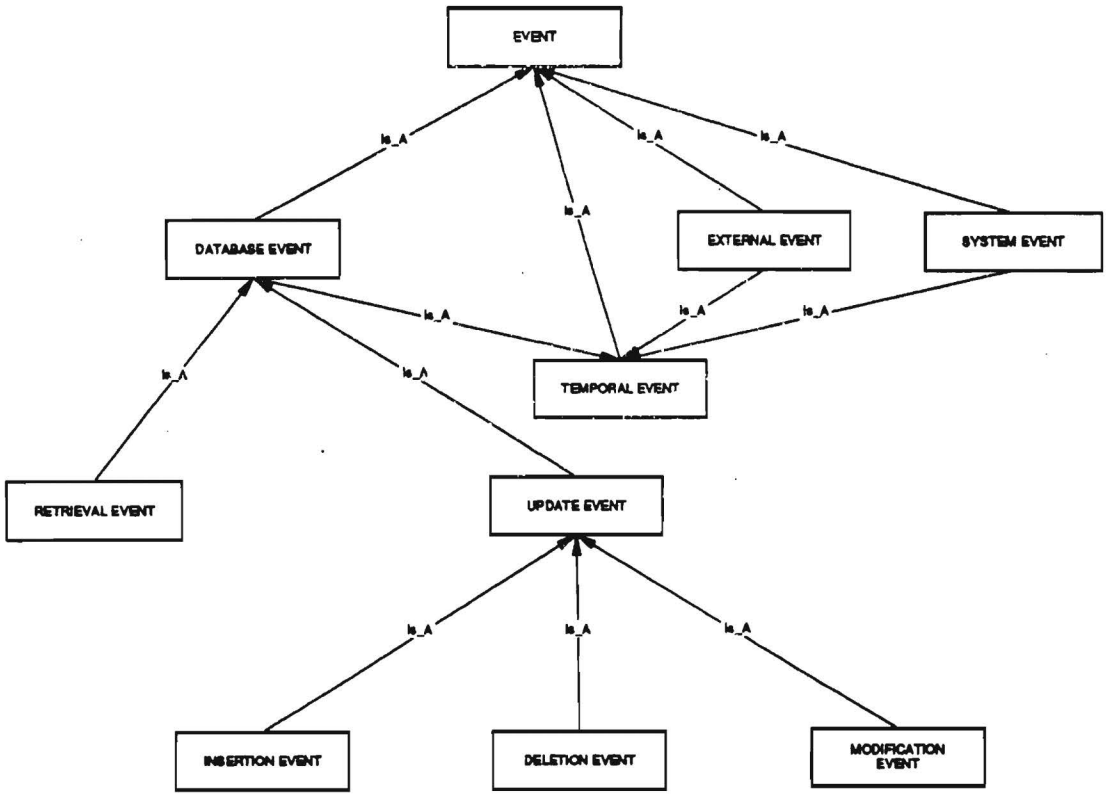

Figure 2.3: Classification of Events

clock events[2].

A database event is the result of a database operation (insertion, deletion, modification, or retrieval) on entities, relationships, or their attributes. In order to allow more flexibility in the specification of rules, we further differentiate an attempted operation from an actually completed operation. This is done by specifying an event immediately before it occurs, which enables us to model the ability to reject an operation before actually executing it. This is essential for modeling the rejection strategy to enforce integrity constraints through active behavior at the conceptual level. At the logical database level, an attempted operation translates into the detection of the event at the time of its occurrence. The operation is suspended until the condition is evaluated; if the condition is satisfied, then the operation that would cause the event is rejected. If the implemented DBMS does not have this capability, the typical solution is to rollback the operation or the entire transaction in which the operation was performed.

[2]By underlying system we mean the operating system and DBMS environment that are potential sources of system events.

External events are signals from the application domain such as "engine stopped" and "reviewer reminded". At the conceptual level, specification of any kind of signal should be possible; the conceptual-to logical translation step, either manual or automatic, should consider the implementation issues of signal detection.

We also distinguish a special type of event called temporal event, which may be a time-constrained signal from an application (external event) or may be related to some time-stamped data object or attribute (database event) or may be a clock event generated by the underlying system (system event). For example, "time-out" may be the result of finding that an expiration date stored in the database has been exceeded or simply an event caused by the system clock. Other examples of temporal events are "every weekday at 5:00 PM", "one month after the occurrence of event E".

We specify an event using the syntax presented in figure 2.4, in which we consider only primitive events and conjunction of primitive events. Some research is being done on other types of composition of events in active DBMSs [Mis91, GJS92], e.g., disjunction, sequence, and closure, but the complexity of the detection of composite events has prevented the development of practical implementations. The most useful type of composition, disjunction of many events, has been implemented in research prototypes (Postgres [SJGP90] and Starburst [WF90]); it is easily modeled in our approach by specifying as many behavior sentences as the number of primitive events in the disjunction, each firing the same list of actions. The notion of conjunction, characterized by the connective "AND" is that of occurrence of events without any specific order; simultaneous occurrence of primitive events is not considered because, by assumption, each event has a unique time of occurrence (and associated detection by the system), i.e., if two events occur simultaneously, either they are the same event or one event subsumes the other. Since we are looking at an immediate practical use of the approach, we leave the incorporation of more complex composition of events in the language for future extensions, which will be quite straightforward at the conceptual level.

In this definition, *event_id* is the unique identification of the event. An *obj_name* is the name of an entity set, a relationship set or a role of an entity in a relationship and *attr_name* is the name of an attribute of object. Objects, roles, attributes, and values are specified in the underlying language for specification of ER schemas, as well as the lexical conventions for "identifier".

Events other than *database_event* (*external_event* and *system_event*) are user-defined signals, possibly with parameters, i.e., event attributes. The definition of a *signal* is implementation dependent; for external events, it will usually consist of a stored procedure invoked from an application program or directly by the user. The following definition syntax is assumed and used as the basis for signal calls in the event and action specification:

signal_definition	::=	CREATE SIGNAL *signal_name* ['(' *formal_parm_list* ')']
formal_parm_list	::=	identifier ':' *value_set*
		[',' identifier ':' *value_set*]*

```
event          ::= [ BEFORE ] event_id ':' event_type
                   [ AND [ BEFORE ] event_id ':' event_type ]*
event_id       ::= identifier
event_type     ::= database_event | external_event | system_event
database_event ::= [attr_name OF] obj_name MODIFIED
                 | obj_name INSERTED
                 | obj_name DELETED
                 | [attr_name OF] obj_name RETRIEVED
obj_name       ::= identifier
attr_name      ::= identifier
external_event ::= signal
system_event   ::= signal
signal         ::= signal_name [ '(' parm_list ')' ]
signal_name    ::= identifier
parm_list      ::= value [ ',' value ]*
```

Figure 2.4: Syntax for Event Specification

where *value_set*, i.e., the allowed set of values for the identifier, is specified in the underlying ER schema. As an example, a timeout mechanism can be specified as:

CREATE SIGNAL timeout(deadline : date),

where "deadline" is the event attribute and may refer to an object attribute or a variable of type "date".

Every *database_event* has a set of pre-defined attributes that it carries to the rules it fires. The event attributes correspond to the attributes of the affected objects, and the notation depends on the type of operation that caused the event. The event attributes may be referenced in the body of the fired rule, i.e., in the specification of conditions and actions.

For an entity, all its attributes are carried by the event, and the following notation is used, where *attr_name* is the name of the attribute as specified in the ER schema:

Type of *database_event*	Predefined attributes
INSERTED	NEW *attr_name*
DELETED	OLD *attr_name*
MODIFIED	NEW *attr_name*, OLD *attr_name*
RETRIEVED	*attr_name*

For a relationship, besides its own attributes, if any, the "NEW" and/or "OLD" attributes of the participating objects are inherited by the relationship and also carried by the event.

2.1.2 Rules

We have defined a rule as the language construct:

rule	::=	*rule_id* ['(' *description* ')'] ['[' *priority_level* ']'] ':' [IF *condition* THEN] *action_list*
rule_id	::=	identifier
description	::=	string
priority_level	::=	identifier

Like *event_id*, *rule_id* is an "identifier", which provides unique identification of the rule; *description* is an optional text that meaningfully describes the rule for documentation purpose; and *priority_level* is an optional identification of the priority of the rule according to a user-defined priority policy. In general, a priority policy defines a number of priority levels: rules with different priority levels are executed in the precedence order of the levels; and rules within the same priority level are executed in some order dictated by the rule selection strategy of the active DBMS. The lower level of priority is the default, and the simplest priority policy is "no policy", where all rules have the same priority level, and the order of execution of a set of fired rules is left to the active DBMS.

A *condition* is a predicate over the state of the database. Its specification in a rule is optional. A *rule* without a *condition* means that the corresponding actions in *action_list* are to be unconditionally performed whenever the associated *event* occurs (is detected). We use a simple syntax for *condition* so that more complex, implementation dependent constructs will need to be defined to completely specify the language. Any computable database predicate in the implementation data model may be used as the condition part of a rule. In general, a predicate may comprise a collection of single predicates connected by "AND" and "OR", possibly negated ("NOT") or quantified by existential quantifiers, and involving aggregate constructs such as "AVERAGE", "SUM", and "COUNT". At the minimum, a predicate must be a single comparison statement between an attribute and a value in the attribute value set or between an attribute and another attribute. We follow the syntax in figure 2.5, where NOT, AND, OR and IN SET_OF connectors are used.

Like objects and attributes, a *value* is specified in the underlying ER schema language, while *rel_operator* depends on the predicates supported by the implementation model (the usual relational operators are $=, >, \geq, <, \leq, \neq$).

A *condition* acts as a guard on *action_list*. If the condition fails, no action will be triggered and the rule execution will fail. For a given rule, the same *condition* guards all actions in the associated *action_list*.

An *action_list* is a sequence of commands that can be database actions, i.e., operations to be performed on data objects and their attributes, or external (user defined) actions such as raising an external event or sending a message. A special type of action, "REJECT_OPERATION", is defined to specify rules to prohibit certain operations, possibly requiring the rollback of the transaction that caused the firing event. Another special type of action, PROPAGATE_OPERATION, allows the specification of a cascaded propagation of

```
condition      ::=  predicate_list
                |   [ NOT ] predicate_list
predicate_list ::=  predicate [ {AND | OR} predicate ]*
predicate      ::=  [ NEW | OLD ] attr_name [OF obj_name]
                    rel_operator value
                |   [ NEW | OLD ] attr_name [OF obj_name]
                    rel_operator [ NEW | OLD ] attr_name [OF obj_name]
                |   [ NEW | OLD ] attr_name [OF obj_name]
                    [NOT] IN SET_OF '(' attr_name [OF obj_name] ')'
```

Figure 2.5: Syntax for Condition Specification

the effect of the event up to the adjacent objects; for example, if an entity is deleted, the deletion is propagated to the relationships which the deleted entity participates in and to the entities that are associated to the deleted entity by "Is_A" and "ID" connections. The syntax for action is shown in figure 2.6.

As mentioned before, the syntax for objects and attributes is defined in the underlying ER schema, and the definition of *predicate* is given in the syntax for *condition*. The lexical conventions for "string" are also defined in the grammar of the ER schema specification language.

Note that "RAISE" is an action that applies to any kind of events that are raised by the execution of the rule. For a *database_action*, a raised event is implicit, so we do not need to specify the fact that a *database_event* is to be raised. For instance, the action "DELETE_RELATIONSHIP Works BETWEEN EMPLOYEE(ssn="123456789") AND PROJECT", which deletes all occurrences of that employee in the relationship Works, implicitly raises the *database_event* "Works DELETED". On the other hand, non-database events such as "Intermediate_Checkpoint" must be explicitly raised by the rule, in the form of a "RAISE" statement. For an action of the type "MESSAGE", no event is raised unless explicitly specified. For example, we may want to raise a specific event "Candidate_Informed" as a consequence of the action "MESSAGE : 'Inform the candidate that his/her application has been denied' ".

In section 2.2 we examine the semantics of the language constructs we propose, based on a semi-formal operational approach.

2.2 Semantics of the Behavior Specification Language

The general semantics of an active database *behavior_sentence* is straightforward and has the following format:

```
action_list      ::=  action [',' action]*
action           ::=  database_action
                   |  external_action
                   |  REJECT_OPERATION
                   |  PROPAGATE_OPERATION '(' db_event_list ')'
database_action  ::=  db_action '(' event_id ')'
db_action        ::=  INSERT_ENTITY obj_name '(' value_list ')'
                   |  INSERT_RELATIONSHIP obj_name ['(' value_list ')']
                                                  BETWEEN rel_obj_list
                   |  DELETE_ENTITY obj_name '(' predicate ')'
                   |  DELETE_RELATIONSHIP obj_name '(' predicate ')'
                   |  DELETE_RELATIONSHIP obj_name ['(' predicate ')']
                                                  BETWEEN rel_obj_list
                   |  MODIFY obj_name '(' predicate ')' SET '(' value_list ')'
value_list       ::=  assignment [',' assignment]*
assignment       ::=  attr_name [OF obj_name] '=' value
rel_obj_list     ::=  rel_obj_pred [AND rel_obj_pred]*
rel_obj_pred     ::=  obj_name ['(' attr_name '=' value ')']
external_action  ::=  RAISE event_id ':' signal_name
                                  [ '(' actual_parm_list ')' ]
                   |  MESSAGE ':' msg [ '(' event_id ':' signal_name
                                  [ '(' actual_parm_list ')' ] ')' ]
actual_parm_list ::=  value [ ',' value ]
msg              ::=  string
db_event_list    ::=  event_id ':' database_event [',' event_id ':' database_event]*
```

Figure 2.6: Syntax for Action Specification

Pre-conditions :

1. *event* is detected. If "BEFORE" is specified in *event*, then the event was detected but has not actually occurred.

2. *condition* is true.

Execution :

The actions in *action_list* are executed.

Post-conditions :

The events resulting from the execution of *action_list* are raised.

Therefore, the semantics of the active database behavior specification language are derived from the operational semantics of the actions in *action_list* of the rules. Since actions are the constructs of the language that produce effects on the state of the database, *event* and *condition* are treated as pre-conditions for the purpose of examining the semantics of a *behavior_sentence*.

2.2.1 Formal Specification of the (ER)^2 Model

In order to derive an operational semantics for the language, i.e. the semantics of the language in terms of the execution of its operations in an abstract computing machine, we need to specify the (ER)^2 model in a formal way. We will use the following notation for the model constructs:

- A : set of attribute names.
- V : set of attribute domains (value sets).
- O : set of names of entity types and relationship types.
- E : set of event identifiers.
- D : set of all possible database events, a database event being the name of an object type in O, followed by the occurred event (INSERTED, DELETED, MODIFIED, RETRIEVED). In the case of MODIFIED, the event may also be preceded by the name of an attribute in A and the keyword OF.
- S : set of signal names.
- R : set of rule identifiers.
- P : set of identifiers of rule priority levels.

Let $domain : A \longrightarrow V$ be a function that maps attribute names to value sets.

Let $event\_name : E \longrightarrow D \cup S$ be a function that maps event identifiers into database events or signal names.

An *entity type descriptor* is a 7-tuple:

$$(ent\_name, ent\_attr\_set, ent\_key\_attr, ent\_notnull\_attr,$$

$$id\_conn, isa\_conn, ent\_inst)$$

where

- $ent\_name \in O$ is the name of the entity type;

- *ent_attr_set* $\subset A$ is the set of attributes of the entity type;
- *ent_key_attr* $\subseteq$ *ent_attr_set* is the set of key attributes of the entity type;
- *ent_notnull_attr* $\subseteq$ *ent_attr_set* is the set of attributes that are not allowed to have NULL values;
- *id_conn* is a pair of sets $(from\_set, to\_set)$ where $from\_set \subset O$ is the (possibly empty) set of names of entity types that are ID-dependent on *ent_name*, and $to\_set \subset O$ is the (possibly empty) set of names of entity types that are identifying owners of *ent_name*;
- *isa_conn* is a pair of sets $(from\_set, to\_set)$ where $from\_set \subset O$ is the (possibly empty) set of names of entity types that are subclasses of *ent_name*, and $to\_set \subset O$ is the (possibly empty) set of names of entity types that are superclasses of *ent_name*;
- *ent_inst* is the set of all instances that belong to the entity type in a given database state. Each instance represents one entity and consists of a value for each attribute in *ent_attr_set* (some of which may be NULL) plus a value for a surrogate-key attribute.

It is required that every entity instance and every relationship instance have a unique, system-generated surrogate-key attribute so that sets of surrogate-key values of any two object types are disjoint. It is also required that the values of attributes in *ent_key_attr* be not NULL.

A *relationship type descriptor* is a 5-tuple:

$$(rel\_name, rel\_attr\_set, rel\_obj\_set, rel\_notnull\_attr, rel\_inst)$$

where

- *rel_name* $\in O$ is the name of the relationship type;
- *rel_attr_set* $\subset A$ is the (possibly empty) set of attributes of the relationship type;
- *rel_obj_set* is a set of triples $(obj\_name, part, card)$ where each *obj_name* $\in O$ is the name of the related object type; *part* $\in$ {"Total", "Partial"} is the participation constraint of *obj_name* in *rel_name*; and *card* $\in$ {"1","M"} is the cardinality constraint of *obj_name* in *rel_name*;
- *rel_notnull_attr* $\subseteq$ *rel_attr_set* is the (possibly empty) set of attributes that are not allowed to have NULL values;
- *rel_inst* is the set of all instances that belong to the relationship type in a given database state. Each instance represents one relationship and consists of a value for each attribute in *rel_attr_set* plus a surrogate-key value for each entity type participating in the relationship. If the relationship involves other relationships, the list of surrogate-key attributes of the participating relationships becomes part of the instance as well.

An *event type descriptor* is a 4-tuple:

$$(event\_id, event\_attr\_set, event\_spec, fired\_rule\_set)$$

where

- $event\_id \in E$ is the identification of the event type;
- $event\_attr\_set \subset A$ is the (possibly empty) set of attributes carried by the event type;
- *event_spec* comprises either a database event in D or the name of the signal (external or system event) in S;
- $fired\_rule\_set \subset R$ is the (possibly empty) set of rules fired by the occurrence of the event type.

A *rule type descriptor* is a 7-tuple:

$$(rule\_id, rule\_attr\_set, priority\_level, firing\_event,$$

$$condition\_spec, action\_list\_spec, raised\_event\_set)$$

where

- $rule\_id \in R$ is the identification of the rule type;
- $rule\_attr\_set \subset A$ is the (possibly empty) set of attributes of the rule type (carried by its firing event);
- $priority\_level \in P$ is the identification of the priority level of the rule type, if any;
- $firing\_event\_set \subset E$ is the (at least singleton) set of event types that fire the rule;
- *condition_spec* is the specification of the condition part of the rule in the active behavior specification language;
- *action_list_spec* is the specification of the actions of the rule in the active behavior specification language;
- *raised_event_set* is the (possibly empty) set of event types raised by the execution of the rule.

A general constraint of the model is that names in O and S and identifiers in E and R be unique throughout the database, as well as names of attributes within $(ER)^2$ object type descriptors.

An $(ER)^2$ schema is a 4-tuple (ENTITY, RELATIONSHIP, EVENT, RULE) where ENTITY is a set of entity type descriptors, RELATIONSHIP is a set of relationship type descriptors, EVENT is a set of event type descriptors, and RULE is a set of rule type descriptors.

2.2.2 Operational Semantics of Actions

As presented in section 2.1, the following actions are specified as part of the *action_list*.

- RAISE *event_id* ':' *signal_name* ['('*actual_parm_list*')']
- MESSAGE ':' *msg* ['('*event_id* ':' *signal_name* ['('*actual_parm_list*')']')']
- REJECT_OPERATION
- PROPAGATE_OPERATION '(' *db_event_list* ')'
- database actions: INSERT_ENTITY, INSERT_RELATIONSHIP, DELETE_ENTITY, DELETE_RELATIONSHIP, and MODIFY.

The operational semantics of each action is defined below in the form of pre-conditions, execution, and post-conditions. This semantics is nested in the semantics of the *behavior_sentence* in which the action appears, i.e., the latter will be, in general, defined as:

Pre-conditions (firing event detected, condition true)
Execution :
pre-conditions, execution, post-conditions for action 1.
pre-conditions, execution, post-conditions for action 2.
. . .
pre-conditions, execution, post-conditions for action N.
Post-conditions (resulting events raised)

It is assumed that the underlying active DBMS in which the actions are executed has the capabilities for suspending, aborting, and rolling back database operations (or the entire transaction in which the operation is performed), as well as an adequate data structure (a list) for keeping track of the detected events. The interesting actions are the external actions ("RAISE" and "MESSAGE") and the special types of database actions ("REJECT_OPERATION" and "PROPAGATE_OPERATION") introduced in the $(ER)^2$ model. The primitive database actions have been well defined in the context of manipulation languages for extended ER models [AH85, CERE88] and transaction specification languages for semantic data models [NB91]. Their operational semantics is based on the relational implementation of the ER schema, i.e., it is based on the operational semantics of the DBMS's DDL/DML, which we do not discuss in this report.

RAISE $e : s(p\_l)$

> Given *event_id* e, *signal_name* s, and *actual_parm_list* p_l, RAISE adds e to the list of detected events; s and the values in p_l must conform to the *event type descriptor* of e.

Pre-conditions:

1. $e \in E$
2. $s \in S$, $event\_name(e) = s$
3. If p_l is specified, let $p\_l =< a_1, a_2, ..., a_n >$
   and *event_attr_set* of $e =< A_1, A_2, ..., A_n >$. Then each value must be in the proper domain, i.e.:
   $a_i \in domain(A_i), 1 \leq i \leq n$

Execution:

Add e to the list of detected events.

Post-conditions:

e is raised.

MESSAGE : $m(e : s(p\_l))$

Given *msg* m, *event_id* e, *signal_name* s, and *actual_parm_list* p_l, MESSAGE outputs m, typically a string of characters. If $e : s(p\_l)$ is specified, it behaves exactly like in RAISE $e : s(p\_l)$.

Pre-conditions:

If $e : s(p\_l)$ is specified, same as 1., 2., 3. for RAISE $e : s(p\_l)$.
Otherwise no pre-condition.

Execution :

Output m. If $e : s(p\_l)$ is specified, add e to the list of detected events.

Post-conditions :

If $e : s(p\_l)$ is specified, then e is raised.
Otherwise no post-condition.

REJECT_OPERATION

If "BEFORE" was specified with the firing event, then the suspended operation that would cause the event is aborted; otherwise, the operation is rolled back.

Pre-conditions:

None

Execution:

If "BEFORE" was specified, then abort the suspended operation. Otherwise, roll back the operation that caused the event.

Post-conditions:

If "BEFORE" was specified, no post-condition.
Otherwise the firing event is raised.

PROPAGATE_OPERATION (e_l)

Selectively propagates the operation that caused the firing event to the adjacent objects listed in (e_l).

Pre-conditions:

1. Let $e\_l =< e_1 : d_1, e_2 : d_2, ..., e_n : d_n >$. Then each *event_id* e_i must be a valid event identifier, i.e.:
   $e_i \in E, 1 \leq i \leq n$
2. Each *database_event* d_i must be a valid database event, i.e.:
   $d_i \in D, 1 \leq i \leq n$
3. Each database event must correspond to its event identifier in the *event type descriptor*, i.e.:
   $d_i = event\_name(e_i), 1 \leq i \leq n$
4. Let o_f be the name of the object in the firing event,
   and $o_i, 1 \leq i \leq n$ the name of the object in the database event d_i. Then
   - If o_f is an entity type, then each o_i must be either an entity type connected to/from o_f by an "ID" or "Is_A" arc, or a relationship type which o_f participates on, i.e.:
     $o_i \in (id\_conn.from\_set \cup id\_conn.to\_set \cup isa\_conn.from\_set \cup isa\_conn.to\_set \cup rel\_conn\_set)$.
   - If o_f is a relationship type, then each o_i must be an object type (entity or relationship) that participates in o_f, i.e.:
     $o\_i = rel\_obj\_set.obj\_name$ for some *obj_name* described in the *rel_obj_set* of o_f.

Execution :

Let o_e be the name of the occurred update operation in the firing event (i.e., the occurred event on o_f). Then the execution semantics of each propagation follows the semantics of the corresponding individual operation, i.e.:

- If o_e = "INSERTED" then execute insert operation on (entity or relationship) $o_i, 1 \leq i \leq n$.

- If o_e = "DELETED" then execute delete operation on (entity or relationship) $o_i, 1 \leq i \leq n$.
- If o_e = "MODIFY" then execute modify operation on (entity or relationship) $o_i, 1 \leq i \leq n$.

Post-conditions:

The post-conditions of the individual propagated operations hold.

2.3 $(ER)^2$ Diagrams

The conceptual schema in the $(ER)^2$ model comprises the usual ER schema plus the specification of active behavior in the form of events and rules. For the conceptual-to-logical mapping step of the database design process, a textual specification of the combined schema in the appropriate language is all that the translation tools need in order to be able to generate the database structure and behavior definition statements in the target DBMS.

The diagrammatic representation extensively used during the conceptual design phase is a graphical tool that helps the database designer in three aspects:

- Communication with the users.
- Automatic generation of the textual specification from the graphical representation.
- Documentation of the design.

We extend these facilities supplied by the ER diagram with a graphical notation for events and rules, to provide the database designer with a means of representing active database behavior along with the structural data constructs.

In an $(ER)^2$ diagram, an event is represented as a circle and a rule as a parallelogram. Directed edges represent connections between events and rules and between events and data objects (entities and relationships). Figure 2.7 shows the representation of a single behavior in terms of the firing event, the rule, and the events raised. The connections between events and rules, and between events and the outside objects are also explicitly shown. In an actual diagram, there is no need to label "Fires" and "Raises" arcs; the connections between events and rules are implicit: an event "fires" rules and a rule "raises" events. The connections "Affects" and "Affected_by" are labeled in an $(ER)^2$ diagram with the type of database event (modification, insertion, deletion, or retrieval). Non-database events (signals), are represented by the events themselves, i.e., they are not connected to data objects.

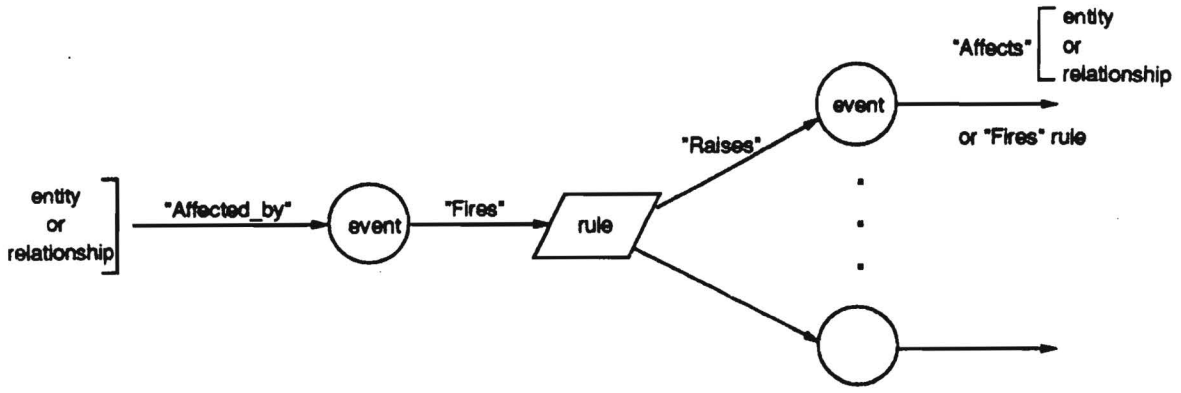

Figure 2.7: Diagrammatic Representation of an Active Behavior

2.3.1 Meta-schema

As we did with the ER model, we can specify the $(ER)^2$ model in a meta-schema and represent it as an $(ER)^2$ diagram, i.e. a meta-$(ER)^2$-diagram of the $(ER)^2$ model itself. This is shown in figure 2.8, where the event and rule objects are integrated in the model with the appropriate notation, and the "ER2_Connection"'s of figure 2.2 are explicitly represented by the links "Affected_by", "Affects", "Fires", "Raises", and by the relationships "Precedes" and "Priority". The external environment (system, applications, and users) is also shown as a potential source and target of events.

This meta-schema is part of the meta-database that stores meta-data about the design process, i.e., the definition of schemas at different levels and their mappings. The meta-database is a self-documentation of the design process, and is an essential source of information for further extensions to the database design methodology.

2.3.2 Example

As an illustration, figure 2.9 shows an $(ER)^2$ diagram of a company's EMPLOYEE-DEPARTMENT-PROJECT database with some events and rules attached to the data objects. The following ER schema is assumed - for simplicity, details such as cardinality ratios ("1","M"), identification dependencies ("ID"), participation constraints ("Total"), and roles ("manager", "employer") are shown only in the diagram, and attributes are specified only in the textual schema:

- EMPLOYEE(ssn, name, job, address, birth_date, status, salary)
- DEPARTMENT(name, location)
- PROJECT(name, budget)

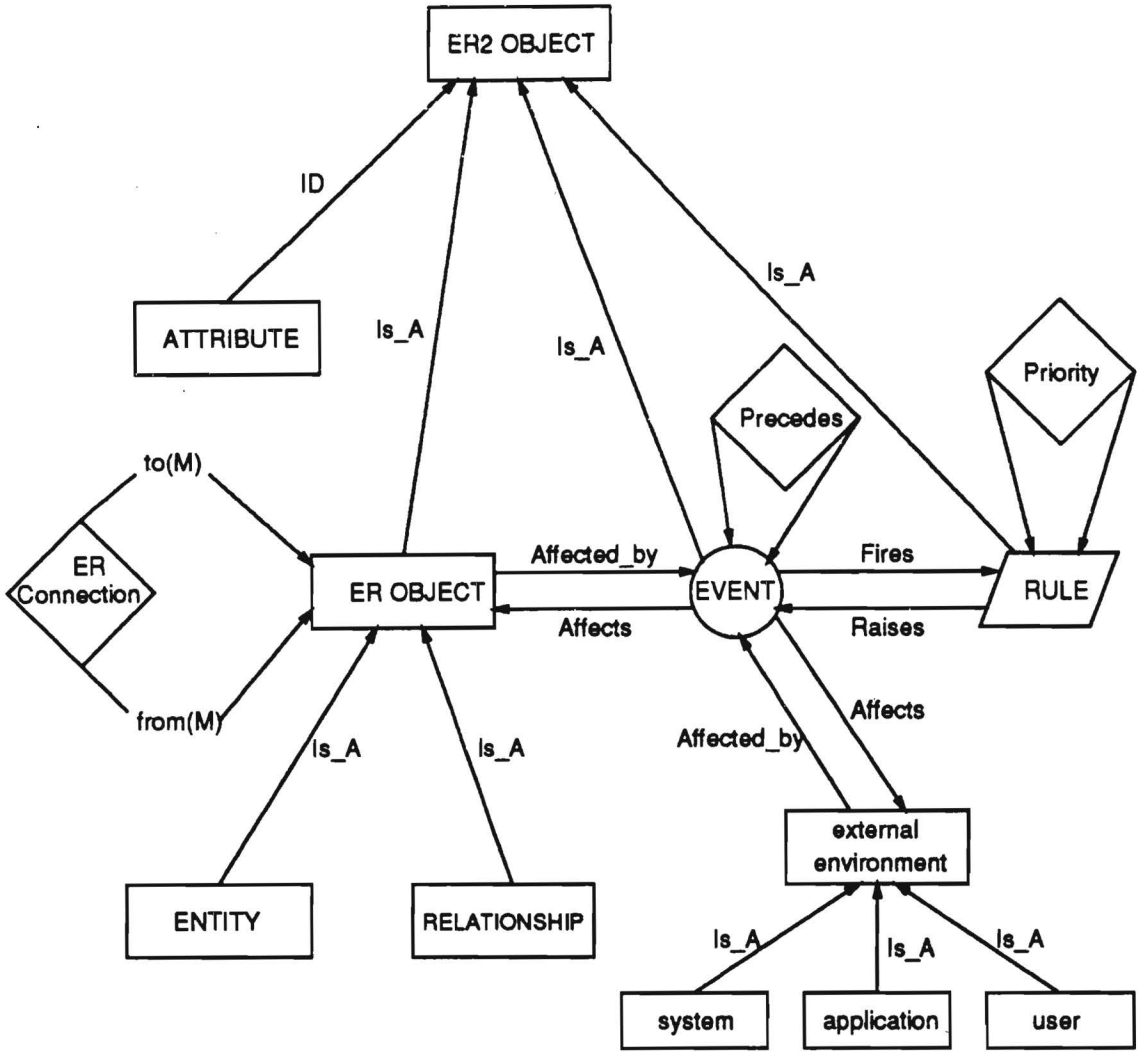

Figure 2.8: Meta-$(ER)^2$-diagram of the $(ER)^2$ Model

- DEPENDENT(EMPLOYEE_ssn, name, birth_date)
- Employed(EMPLOYEE_ssn, DEPARTMENT_name)
- Manages(EMPLOYEE_ssn, DEPARTMENT_name)
- Works(EMPLOYEE_ssn, PROJECT_name, start_date, hours_week)

The following behavioral sentences are specified in terms of the events and rules represented in figure 2.9:

WHEN e1 : PROJECT MODIFIED

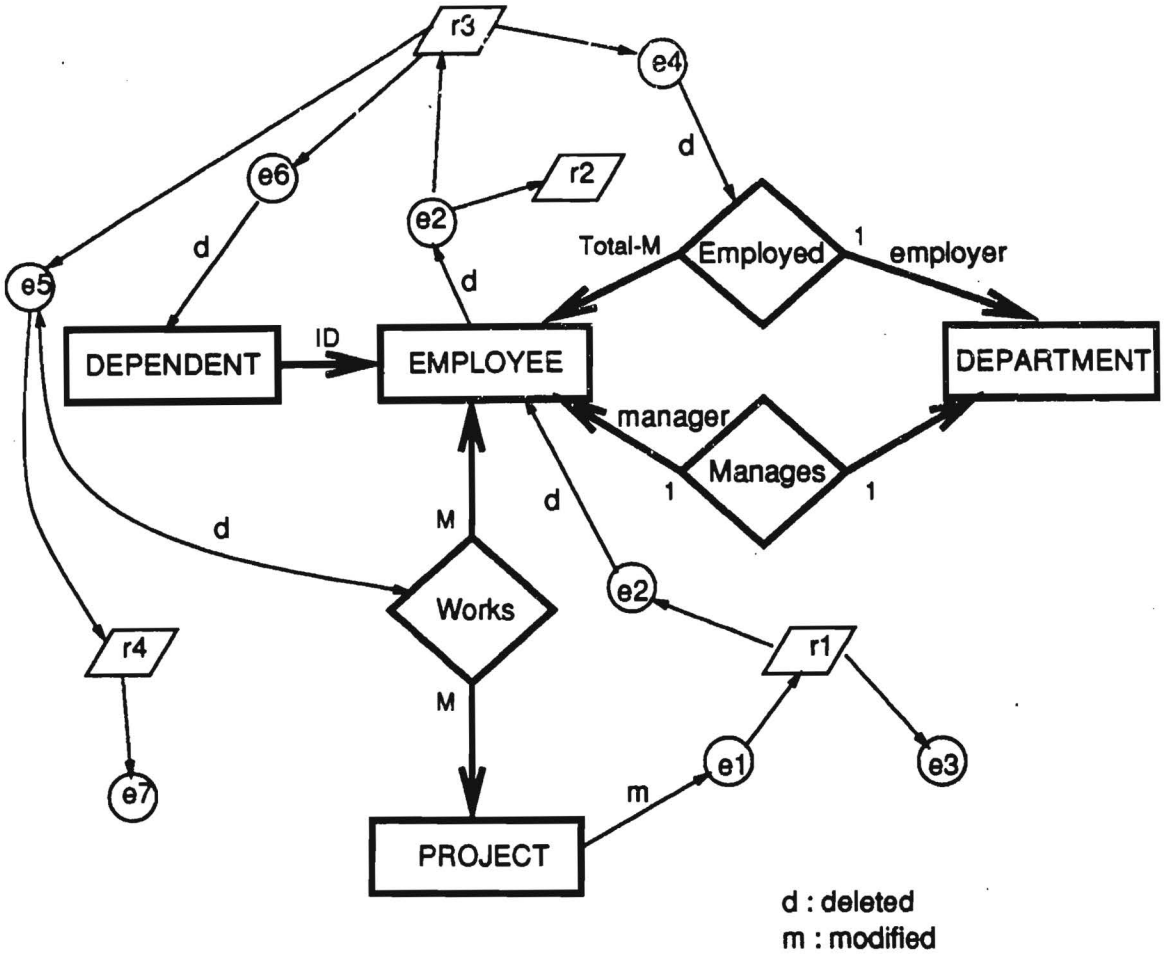

Figure 2.9: $(ER)^2$ Diagram of a COMPANY Database

FIRE r1 ("Policy for budget reduction") :
IF NEW budget < OLD budget
THEN DELETE_ENTITY EMPLOYEE (ssn = OLD EMPLOYEE_ssn,
status = "temporary") (e2),
RAISE e3 : salary_review.

WHEN e2 : EMPLOYEE DELETED
FIRE r2 ("Restriction to firing engineers") :
IF OLD job = "engineer"
THEN MESSAGE : "Employee is an engineer, deletion rejected",

REJECT_OPERATION.

WHEN e2 : EMPLOYEE DELETED
FIRE r3 ("Cascaded deletion of temporary employees") :
IF OLD status = "temporary"
THEN PROPAGATE_OPERATION (e4 : Employed DELETED,
e5 : Works DELETED, e6 : DEPENDENT DELETED).

WHEN e5 : Works DELETED
FIRE r4 ("Warning message to project manager") :
MESSAGE : "Inform change on employee assignment to project manager"
(e7 : manager_warning).

The $(ER)^2$ diagram represents the active database behavior in the form of events and rules and their interaction with data objects. To avoid cluttering the diagrammatic representation, we chose to keep the specification of events, conditions, and actions apart from the diagram, using textual description. The same user interface design technique is adopted by most of the current ER diagramming tools, where the attributes are specified in pop-up windows that are displayed when the corresponding object symbols are clicked. This technique keeps the diagram simple and easy to read, without loss of information. In addition, the use of different line styles and colors for structural and behavioral constructs would help making the diagram more readable.

As shown in the above example, the $(ER)^2$ model can capture a variety of constraints and situation/action behaviors, such as a high-level organizational policy (rule r1), a restrictive prescription (rule r2), the enforcement of an integrity constraint (rule r3), or a database event alerter (rule r4). Potentially, this framework can represent any application-relevant behavior that can be managed by an active DBMS. In addition, this representation can be easily adapted to data abstraction extensions to the ER model such as generalization/specialization and aggregation. As pointed out before, those extensions do not disturb the $(ER)^2$ framework because of the orthogonality of the added dimension.

2.4 Mapping of $(ER)^2$ Specification into DBMS Constructs

2.4.1 Meta-database of the Schema Translation

During the traditional schema translation process from conceptual to logical level, the translation tools acquire knowledge about the conceptual schema from the input file, apply the mapping procedures, and generate the logical schema in the target DBMS. This meta-information about the schemas and their mapping is a valuable resource that must be stored in a meta-database for further use.

The most visible use of the meta-database is for self-documentation of the design and possible use as a basis for schema evolution management [MR90]. In a full fledged CASE

environment, the meta-database can be combined with additional meta-information about the process design to form a project database.

The meta-database is also useful for extensions to the query capability of the database system. For instance, graphical languages for querying the conceptual schema are more concise and adequate for naive database users than the query languages provided by the DBMS, e.g., SQL. The information stored in the meta-database is fundamental for the translation of queries in the conceptual schema to actual queries in the DBMS language. This idea can be extended to querying multiple, heterogeneous database systems, in which various meta-databases would be integrated for assisting the different conceptual-to-logical query transformations. Some highly specialized database applications, such as scientific applications, may require a specific query language that must also use the meta-database for translation [MLM+92].

In our work, we use the meta-database as the central repository of schema design information to incorporate active behavior translation in the database design process. Figure 2.10 shows the types of meta-information needed for this purpose:

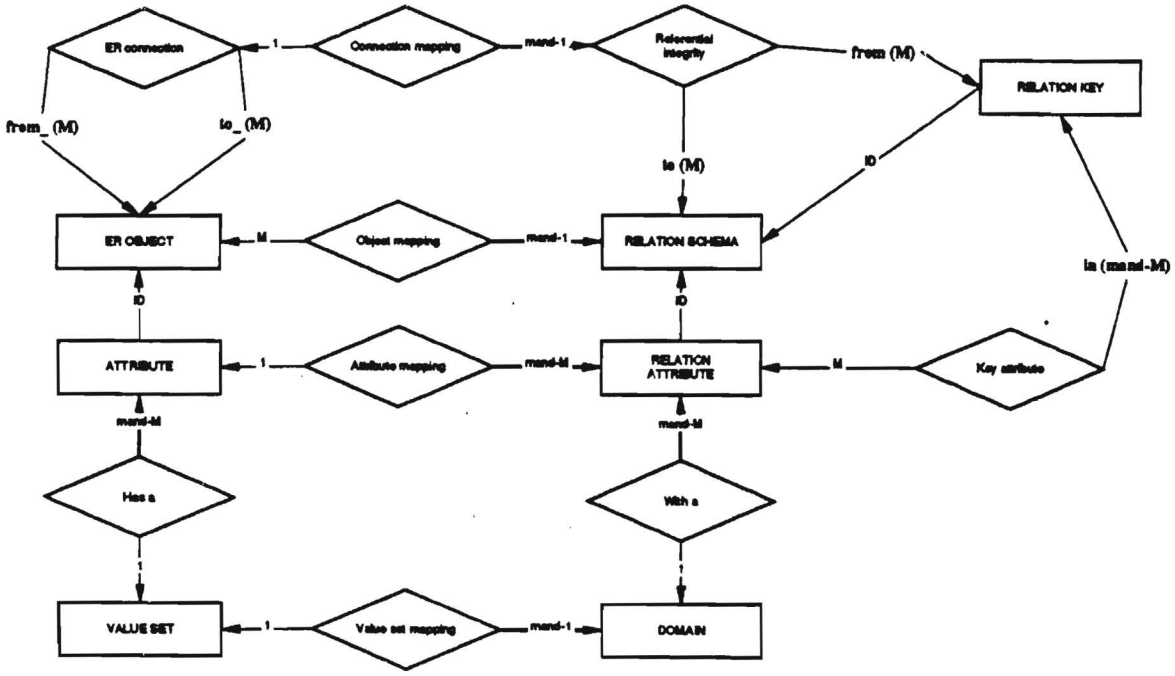

Figure 2.10: Meta-database: ER schema, Relational schema, and their Mapping (LBL tool set)

- ER objects (Entity sets and Relationship sets) and their mapping into relation schemas.

- ER object connections (arcs in the ER diagram) and their mapping into referential integrity constructs supported by the DBMS.
- Object attributes and their mapping into relational attributes, along with information about key attributes.
- Value sets and their mapping into the domains supported by the target DBMS. Usually schema translation tools are implemented in such a way that the value sets of attributes in the ER schema are tailored to the domains in the target DBMS; hence, the value set-to-domain mapping is a one-to-one mapping. When the tools support more than one target DBMS, different value set options are available for the specification of attributes.

2.4.2 Active DBMS Language Constructs

Although a standardization effort is being done in the area of data definition and manipulation languages, there are significant differences between the various commercial DBMSs. In particular, the syntax of the active database constructs varies significantly. Here we consider the constructs currently present in commercial relational DBMSs; they reflect the development of research prototypes, from System R to Starburst and Postgres.

Three types of active database constructs have been implemented as part of the data definition language of relational DBMSs: triggers, rules, and exception handlers. The functionality of triggers and rules is similar in practice, since both are general mechanisms to specify active behavior in the form of situation/action rules, database triggers, and event alerters.

Triggers are present in the proposed standards SQL2 and SQL3 [Mel90] and implemented in Sybase [SYB87], Oracle [Kos92], and Interbase [INT90]. The syntax varies from product to product; the following is the proposed syntax in SQL2/SQL3:

```
CREATE TRIGGER trigger_name
        { AFTER | BEFORE }
        { INSERT | DELETE | UPDATE [OF column_list] }
                    ON table_name
        [ REFERENCING
                    { OLD [ AS ] correlation_name
                                [ NEW [ AS ] correlation_name ] } |
                    { NEW [ AS ] correlation_name
                                [ OLD [ AS ] correlation_name ] } ]
        [ WHEN search_condition ]
                    triggered_SQL_statement_list
                    [ FOR EACH { ROW | STATEMENT } ]
```

where *trigger_name*, *table_name* and *correlation_name* are respectively identifiers for the trigger, the tables involved in the trigger, and the correlated tables used as aliases for the transition tables containing old and new values of the affected tuples; *column_list* is a list of attribute

identifiers; and *search_condition* and *triggered_SQL_statement_list* are predicates and operations specified in the underlying SQL data manipulation language.

Rules are the basic constructs in Ingres [ING90] and RDB/VAX [DEC89]. The following is the syntax in the Knowledge Management Extension of Ingres; it combines rules with stored procedures.

```
CREATE RULE rule_name
        AFTER
        { INSERT INTO | DELETE FROM |
                        UPDATE ON | UPDATE column_name OF }
                        table_name
        [ REFERENCING [ OLD AS correlation_name ]
                        [ NEW AS correlation_name ] ]
        [ WHERE qualification ]
        EXECUTE PROCEDURE procedure_name '(' procedure_parameters ')'

CREATE PROCEDURE procedure_name '(' parameter_list ')'
        AS [ declare_section ]
        BEGIN
                        statement_list
        END
```

Here also *rule_name*, *column_name*, *table_name*, *correlation_name*, and *procedure_name* are identifiers; and *qualification* is a predicate over the affected tables. A procedure is a mechanism with the full power of a database programming language containing parameters, a *declare_section* for declaration of local variables, and a *statement_list* that includes database operations, control statements (IF, WHILE, RETURN), assignment statements, MESSAGE and RAISE ERROR statements.

Although it lacks many desirable features such as detection of events "after" and "before" they occur, priority information, retrieval events, and arbitrary procedures as events to emulate signals, the rule/procedure construct of Ingres has the advantage of modularity. Potentially, this approach could be extended to incorporate all these features and could be used in any SQL extension with stored procedures and rules or triggers, such as in SQL2/SQL3, Sybase, and Oracle. In the following mapping algorithms, we use this construct as the general language construct of the target active DBMS.

The last type of construct present in current active DBMSs is the exception handler, a very limited mechanism to deal with signals. The general construct is of the form:

```
WHENEVER SQL_signal exception_action,
```

where *SQL_signal* can be any of a few error codes originating from embedded SQL statements, and *exception_action* is basically a message or a jump/call to an exception handling procedure written in the host language. Although the functionality is present for a restricted set of

system signals, we will not deal with this mechanism because of its limitation (the *SQL_signal* basically indicates the occurrence of an error in the execution of SQL commands embedded in programs written in a host language).

2.4.3 Mapping Process

Recall that the specification of active behavior in the $(ER)^2$ model is a list of *behavior_sentences*, each one defined as:

---

behavior_sentence ::= WHEN *event* FIRE *rule* '.'
event ::= *event_id* ':' *event_type*
rule ::= *rule_id* ['(' *description* ')'] ['[' *priority_level* ']'] ':'
[IF *condition* THEN] *action_list*

We assume the following syntax for rule definition in the target DBMS, which is more general than that of Ingres ; the procedure definition is the same as in Ingres:

CREATE RULE *rule_name* [*description*] [*priority_level*]
[BEFORE | AFTER]
{ *signal_name* '(' *signal_parameters* ')' |
{ INSERT INTO | DELETE FROM |
UPDATE ON | UPDATE *column_name* OF |
RETRIEVE ON | RETRIEVE *column_name* OF } *table_name* }
EXECUTE PROCEDURE *procedure_name* '(' *procedure_parameters* ')'

The following meta-database look-up functions are defined for assisting the mapping process:

get_obj_map(*obj_name*) returns the table name to which *obj_name* was mapped.

get_attr_map(*attr_name*) returns the column name to which *attr_name* was mapped.

get_conn(*obj_name*) returns the list of objects connected to *obj_name* in the ER schema, along with the type of each connection.

get_attr_list(*obj_name*) returns the list of attributes of *obj_name*.

get_attr_list_map(*obj_name*) returns the list of column names to which the list of attributes of *obj_name* was mapped. In an actual implementation, this function is performed through a get_attr_list(*obj_name*) followed by a sequence of get_attr_map(*attr_name*).

get_value_set_map(*value_set*) returns the domain name to which *value_set* was mapped.

Not surprisingly, with the information provided by the meta-database the mapping becomes straightforward. The mapping process proceeds as follows.

1. Input: a list of *behavior_sentences*.
   Each *behavior_sentence* generates a pair (rule definition, procedure definition), as described below.

2. Each *obj_name* referred in the *event*, *condition*, or *action_list* parts of the *behavior_sentence* is mapped into a *table_name* using get_obj_map(*obj_name*).

3. Each *attr_name* is mapped into a *column_name* using get_attr_map(*attr_name*).

4. Each *value* referred in the *condition* or *action_list* parts of the *behavior_sentence* is mapped into a value in the corresponding domain obtained by get_value_set_map (*value_set*).

5. Rule definition:

   - Output:
     CREATE RULE *rule_id* [*description*] [*priority_level*]
     { BEFORE | AFTER } *event*
     EXECUTE PROCEDURE *procedure_name* (*procedure_parameters*);
   - *rule_id* maps directly into *rule_name* unless corrections are needed to meet naming conventions for identifiers.
   - *description* maps into a string of characters for documentation purpose.
   - *priority_level* maps into an implementation-dependent definition of priority (e.g. an integer sequence number).
   - *event* maps into the corresponding language construct:
     - Case *database_event*:
       * *attr_name* OF *obj_name* MODIFIED ⟶ UPDATE *column_name* OF *table_name*
       * *obj_name* MODIFIED ⟶ UPDATE ON *table_name*
       * *obj_name* INSERTED ⟶ INSERT INTO *table_name*
       * *obj_name* DELETED ⟶ DELETE FROM *table_name*
       * *attr_name* OF *obj_name* RETRIEVED ⟶ RETRIEVE *column_name* OF *table_name*
       * *obj_name* RETRIEVED ⟶ RETRIEVE ON *table_name*
     - Case *external_event* or *system_event*:
       * The translation into *signal_name* [(*signal_parameters*)] will depend on the implementation of signals.
   - The translation of *procedure_name* (*procedure_parameters*) will depend on the procedure definition, as explained below.

6. Procedure definition:

- Output:
  CREATE PROCEDURE *procedure_name* (*parameter_list*)
  AS *declare_section*
  BEGIN
  statement_list
  END;
- *procedure_name* is mapped into proc_*rule_id* to associate the identification of the procedure to the rule that calls it.
- *parameter_list* corresponds to the attributes of *event*:
  - Case *database_event* : the parameters are the predefined event attributes (see the table at the end of section 2.1.1) mapped into *column_names* using get_attr_map(*attr_name*). Each *column_name* is prefixed by "o_" (for old) or "n_" (for new) if the event was an update event, and followed by the domain corresponding to the *value_set* of *attr_name*. For example, suppose attribute "DEPARTMENT_name" of object "Employed" with *value_set* "varchar" is mapped into column "dname" of relation "EMPLOYEE" with domain "varchar"; then if the event is "Employed DELETED", the corresponding procedure parameter will be "o_dname varchar"; if the event is "Employed INSERTED", it will be "n_dname varchar"; and if the event is "Employed MODIFIED", it will be "o_dname varchar, n_dname varchar". Also, in this example, note that *table_name* = get_obj_map("Employed") = "EMPLOYEE".
  - Case *external_event* or *system_event*: each parameter is a user-defined pair "identifier : *value_set*" mapped into the corresponding "identifier domain" in *parameter_list*. Every identifier maps into an identical name unless corrections are needed to meet naming conventions of the target DBMS.
  - In any case, the actual *procedure_parameters* in the EXECUTE PROCEDURE statement inside the rule definition will be a list of pairs "formal_parameter_name = value", one for each parameter in *parameter_list*. A "formal_parameter_name" is as defined above and "value" is either the *column_name* prefixed by the qualification keywords "new." or "old." for database events or a user-specified value mapped into the corresponding domain for non-database events. For example, the actual parameter corresponding to the formal parameter "n_dname varchar" will be "n_dname = new.dname".
- The *declare_section* contains declarations of variables that are locally referenced by the procedure. The following variables will be used in the definition of procedures invoked by rules:
  - "message string" to keep the text of *msg* specified with the *action* "MESSAGE : msg".
  - "counter integer" to keep the number of tuples in a table that is used for checking the existence of tuples satisfying some condition.

- The *statement_list* will contain the statements corresponding to the *condition* and *action_list* parts of the *rule*.
  - For each *predicate* in *condition*:
    * Case [NEW | OLD] *attr_name* [OF *obj_name*] *rel_operator value*
      $\longrightarrow$ IF [*table_name*].[n_ | o_]*column_name rel_operator value*
      where *rel_operator* and *value* are mapped into the corresponding operator and value in the target DBMS language.
    * [NEW | OLD] $attr\_name_1$ [OF obj_name_1] *rel_operator* [NEW | OLD] $attr\_name_2$ [OF obj_name_2]
      $\longrightarrow$ IF [$table\_name_1$].[n_ | o_]$column\_name_1$ *rel_operator* [$table\_name_2$].[n_ | o_]$column\_name_2$
    * [NEW | OLD] $attr\_name_1$ [OF obj_name_1] [NOT] IN SET_OF ($attr\_name_2$ [OF obj_name_2])
      $\longrightarrow$ IF [$table\_name_1$].[n_ | o_]$column\_name_1$ [NOT] IN (SELECT $column\_name_2$ FROM $table\_name_2$)
      If obj_name_i is omitted, then $table\_name_i$ is the same as the *table_name* corresponding to the *obj_name* in the firing event.

  - For each *action* in *action_list*:
    * Case INSERT_ENTITY *obj_name* (*value_list*)
      $\longrightarrow$ INSERT INTO *table_name* VALUES ($column\_name_i = value_i$)
      for $1 \leq i \leq n$, assuming n is the number of columns in *table_name*.
    * Case INSERT_RELATIONSHIP *obj_name* [(*value_list*)] BETWEEN *rel_obj_list*
      $\longrightarrow$ INSERT INTO *table_name* VALUES ($column\_name_i = value_i$)
      for $1 \leq i \leq n$, where $column\_name_i$ includes the foreign keys of the related tables mapped from *rel_obj_list*.
    * Case DELETE_ENTITY *obj_name* (*predicate*)
      $\longrightarrow$ DELETE FROM *table_name* WHERE *predicate*
      where the mapping of *predicate* is similar to that in *condition* above (without the IF clause).
    * Case DELETE_RELATIONSHIP *obj_name* (*predicate*)
      $\longrightarrow$ DELETE FROM *table_name* WHERE *predicate*
      the same as the previous case.
    * Case DELETE_RELATIONSHIP *obj_name* [(*predicate*)] BETWEEN *rel_obj_list*
      $\longrightarrow$ DELETE FROM *table_name* WHERE $predicate_i$
      where $predicate_i$ includes the equality condition on the foreign keys of the related tables mapped from *rel_obj_list*.
    * Case MODIFY *obj_name* (*predicate*) SET (*value_list*)
      $\longrightarrow$ UPDATE *table_name* SET ($column\_name_i = value_i$) WHERE *pred-*

icate
for $1 \leq i \leq n$, where n is the number of modified attributes of *obj_name*; *predicate* is mapped like in *condition*.

* Case REJECT_OPERATION
  ⟶ ROLLBACK [*operation*]
  where *operation* is the database operation associated with the firing event; *operation* is left optional for the case in which the target DBMS does not support rollback at operation level. Alternatively, if rollback inside rules is not supported at all (e.g. in the current version of Ingres), REJECT_OPERATION maps into RAISE ERROR *error#* : *message*, where *message* is a mandatory warning to the user or application that originated the firing event.
* Case PROPAGATE_OPERATION (*db_event_list*)
  generates a sequence of operations of the same type as in the firing event. The propagation is performed on the objects explicitly specified in *db_event_list*, that must be adjacent to the firing event object in the ER schema. As shown before (section 2.2.2), the propagation does not cascade automatically to other, non-adjacent objects, unless additional rules are specified that deal with the new propagations. The tool checks the adjacency by invoking the meta-database function get_conn(*obj_name*) where *obj_name* is the name of the event object. If the specified list is correct, the propagated operations are generated:
  ⟶ INSERT INTO *table_name* VALUES *value_list*
  ⟶ DELETE FROM *table_name* WHERE *predicate*
  ⟶ UPDATE *table_name* SET *value_list* WHERE *predicate*, where *table_name* is obtained using get_obj_map (*adjacent_obj_name*), while *value_list* and *predicate* are derived from the firing event and the condition part of the rule.
* Case RAISE *event_id* : *signal_name* [(*actual_parm_list*)]
  ⟶ EXECUTE PROCEDURE *signal_name*[(*actual_parm_list*)]
  assuming that a signal is implemented as a stored procedure (currently no commercial DBMS has such functionality, although a few have stored procedures).
* Case MESSAGE : *msg* [(*event_id* : *signal_name* [(*actual_parm_list*)])]
  ⟶ *message* = *msg*; MESSAGE : *message*;
  where *message* is the local variable keeping the text. If *event_id* : *signal_name* [(*actual_parm_list*)] is specified, a separate procedure containing the MESSAGE statement is created to emulate the signal.

As an example, the *behavior_sentence*

WHEN e01 : EMPLOYEE DELETED

```
FIRE r01 : PROPAGATE_OPERATION ( e02 : DEPENDENT DELETED,
                e03 : Employed DELETED, e04 : Manages DELETED,
                e05 : Works DELETED ),
           MESSAGE : 'All references to EMPLOYEE being deleted'.
```

generates the following DBMS procedure/rule definitions:

```
CREATE PROCEDURE proc_r01
      (o_salary float, o_status varchar, o_birth_date date, o_address varchar,
      o_job varchar, o_name varchar, o_ssn char(9), o_DEPARTMENT_name
      varchar, o_Employed_DEPARTMENT_name varchar)
AS DECLARE message VARCHAR NOT NULL; counter INTEGER;
BEGIN
      SELECT COUNT(*) INTO counter FROM Works
            WHERE EMPLOYEE_ssn = :o_ssn;
      IF counter > 0
            THEN DELETE FROM Works
                  WHERE EMPLOYEE_ssn = :o_ssn;
      ENDIF;
      SELECT COUNT(*) INTO counter FROM DEPENDENT
            WHERE EMPLOYEE_ssn = :oo_ssn;
      IF counter > 0
            THEN DELETE FROM DEPENDENT
                  WHERE EMPLOYEE_ssn = :o_ssn;
      ENDIF;
      message = 'All references to EMPLOYEE being deleted';
      MESSAGE :message;
END;

CREATE RULE r01 AFTER DELETE FROM EMPLOYEE
      EXECUTE PROCEDURE proc_r01
      (o_salary = old.salary, o_status = old.status, o_birth_date = old.birth_date,
      o_address = old.address, o_job = old.job, o_name = old.name,
      o_ssn = old.ssn, o_DEPARTMENT_name = old.DEPARTMENT_name,
      o_Employed_DEPARTMENT_name = old.Employed_DEPARTMENT_name);
```

This example also illustrates the mapping of different objects into the same table (EMPLOYEE, Employed, Manages into EMPLOYEE), as well as attributes to columns and how the active behavior mapping process takes advantage of the information on the schema translation to generate the definition of rules and procedures in the target DBMS.

The mapping process described above is direct, i.e., it translates the behavior definition into the specification of procedures and rules in the target DBMS without attempting to

generate the most efficient specification. Like other SQL constructs, rules and procedures can be correctly specified in various ways; optimization issues have to be addressed after the translation, with the generated specification as the starting point.

An example of possible further optimization is the evaluation of set-oriented predicates. The condition part of a rule can be split into two sets of predicates: one referring to the object affected by the firing event, and the other containing the remaining predicates. The first set can be specified in the WHERE *qualification* clause of the rule, outside the procedure that evaluates the remaining predicates. This splitting restricts the amount of data passed to the procedure, making its execution more efficient.

2.5 Summary

In summary, in this section we described the $(ER)^2$ model. First, we introduced events and rules as objects of the model and presented a syntax for the specification of active database behavior using events and rules. Then we described the model using the ER formalism and derived a semantics of the active behavior specification language based on the operational semantics of the action part of the rules. We introduced $(ER)^2$ diagrams, in which events and rules are represented along with entities and relationships, as a graphical tool to help the database designer in the specification of active behavior. Finally we described the mapping of the active constructs in the $(ER)^2$ schema specification into DBMS rules and stored procedures. The mapping algorithms are intended for use by a translation tool that automatically generates the executable DBMS language statements corresponding to the active behavior specified in the $(ER)^2$ schema.

Section 3

CONSTRAINT MODELING AS ACTIVE DATABASE BEHAVIOR

When dealing with database constraints within the ER model, the main issue is the mismatch between constraints specification and enforcement: while specification of constraints has a declarative nature, their enforcement requires procedural language constructs. The relational model has a few inherent constraints such as the key constraint and referential integrity constraints based on foreign keys. However, there is no such general mechanism like a constraint enforcement subsystem in a DBMS that automatically enforces semantic integrity of the database without the need of writing constraint-checking statements in the transactions. In section 3.1 we show that the $(ER)^2$ approach combined with the active database language constructs in the DBMS is also useful for the specification and translation of integrity constraints. This is a typical internal application of the active database paradigm that benefits the database system services. In particular, we show in section 3.2 that the inherent and implicit constraints of the model, also known as invariant properties, can be mapped into $(ER)^2$ schema specification as meta-behaviors, and translated into triggers and event alerters that will enforce their preservation. In section 3.3, we show that dynamic constraints, which require consistency checking of database state transitions as opposed to individual states, are usually better specified directly in the active behavior specification language.

3.1 Integrity Constraints

Two solutions are used to enforce integrity constraints during update operations on the database. The first solution is to prevent the execution of constraint-violating operations (rejection strategy), and the second solution is to permit all correct operations and propagate them to related objects, if necessary for preserving the integrity of the database (propagation strategy).

Definition: A constraint is a predicate that must be satisfied at all time during the existence of the database.

Basically, a constraint is a predicate similar to the type used to specify the condition part of a rule. We use the following syntax:

constraint ::= *constraint_id* ':' *predicate_list*
constraint_id ::= identifier

where *constraint_id* is a unique identification of the constraint and *predicate_list* is the same as defined for the *condition* part of a *rule*.

For example,

c1 : salary of EMPLOYEE $\geq$ 10,000

is a simple constraint that restricts the values of the salaries of employees in a way that is not usually representable in the conceptual schema. Potential violating events (i.e., the operations that cause the events) are the insertion of an employee and the modification of the salary of an employee. Hence some enforcement action, either a rejection or a correction, must be performed when such events are detected and if the outcome is an invalid salary. This behavior is exactly the active database behavior in the form of events and rules.

As another example,

c2 : salary of EMPLOYEE < salary of manager AND
DEPARTMENT_name of manager = DEPARTMENT_name of employer AND
EMPLOYEE_ssn of Employed = ssn of EMPLOYEE

is a constraint that restricts the salaries of employees to be less than the salary of the manager of the department in which they are employed. The potential constraint-violating events in this case are any modification on salary or insertion of an employee with a salary and any changes on the "Manages" or "Employed" relationship caused by insertion or modification.

The active database behavior derived from the specification of static constraints as in these examples requires the following information to be completely defined:

- The potential constraint-violating events, each of which will become the *event* part of an active behavior.
- The invalid new database state generated by the violating events, which will become the *condition* part of the *rule*.
- The list of constraint-enforcing actions to be performed when each violating event is detected and if the new database state is invalid. This list will become the *action_list* of the *rule*.

In other words, each potential constraint-violating event will derive an active database behavior with the following general format that can be generated with the help of a translation tool:

WHEN *event_id*	:	*violating_event*
FIRE *rule_id*	:	IF *invalid_database_state*
		THEN *enforcing_action_list*

where *event_id* and *rule_id* are tool-generated identifications of the event and the rule.

This derivation is not completely automatable, because the *enforcing_action_list* depends on the enforcement strategy adopted for the constraint; also, specification of user-defined messages may be desirable as part of the action list. Thus a tool to assist the constraint-to-behavior mapping process would have an automated step, i.e., the generation of *violating_events* and the *invalid_database_state* plus an interactive step in which the user (database designer) specifies the *enforcing_action_list*. Figure 3.1 illustrates this process; the figure also shows another possibly automated step, the translation of active behavior into DML language constructs in a target DBMS.

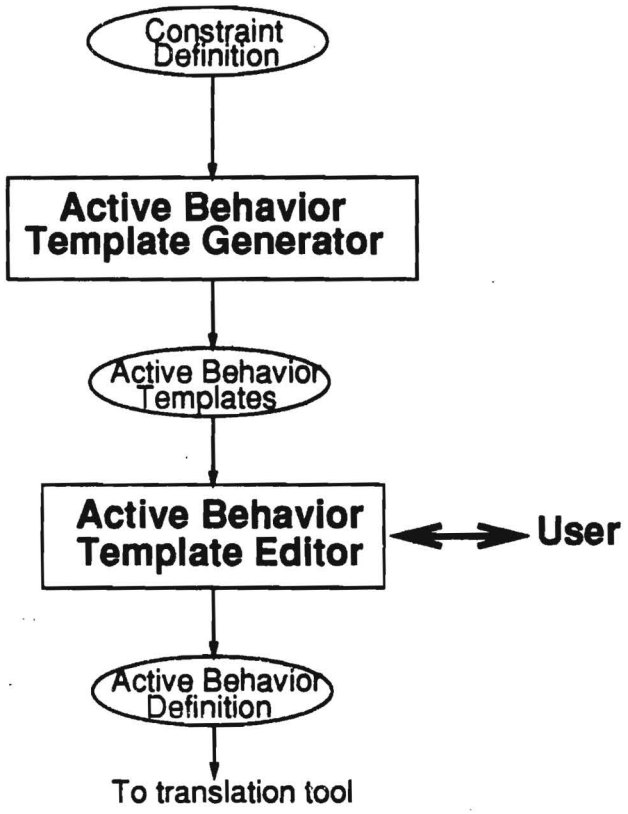

Figure 3.1: Interactive Framework for Active Behavior Derivation from Constraints

In this framework, the derivation of an *invalid_database_state* is straightforward: the database will be in an invalid state if the *predicate_list* of the constraint is false. As a consequence, the *condition* part becomes the negation of the *predicate_list*

invalid_database_state ::= NOT (*predicate_list*).

The set of *violating_events* is derived directly from the syntactic analysis of the constraint. This issue is thoroughly examined in [CW90] in the context of an SQL-based constraint language that includes aggregate functions and set operations and rules in the Starburst prototype DBMS. In our opinion, there is a trade-off between the expressiveness of the constraint language and the simplicity of its declarative semantics. The constraint-to-behavior mapping is useful because behavior is procedural, thus more difficult to understand and program. If the constraint language becomes as complex as the behavior language, the mapping will not be useful anymore, because the behavior will have to be specified to enforce the constraints anyway.

The following mapping rules are used to derive the *violating_events* from the specification of a *constraint*:

1. For each *attr_name* that appears in a *predicate*, modification of the corresponding attribute and insertion of the owner of the attribute (*obj_name*) are potential *violating_events*.

2. For each *obj_name* that appears in a *predicate*, modification and insertion of the corresponding object are potential *violating_events*.

In the above first example, constraint "c1" generates the following templates of active behavior:

WHEN [BEFORE] e1-c1 : salary OF EMPLOYEE MODIFIED
FIRE r1-c1 : IF NOT (salary OF EMPLOYEE $\geq$ 10,000)
THEN *enforcing_action_list.*

WHEN [BEFORE] e2-c1 : EMPLOYEE INSERTED
FIRE r2-c1 : IF NOT (salary OF EMPLOYEE $\geq$ 10,000)
THEN *enforcing_action_list.*

The BEFORE option is left open because it depends on what the user wants to specify in the action list. A rejection strategy will require the event being detected before it occurs, or, alternatively, the rollback of the operation after the event has occurred. A propagation strategy will take effect after the event occurs.

In the second example, the templates generated for constraint "c2" are:

WHEN [BEFORE] e1-c2 : salary OF EMPLOYEE MODIFIED
FIRE r1-c2 : IF NOT (salary of EMPLOYEE < salary of manager AND

DEPARTMENT_name of manager = DEPARTMENT_name of employer AND
EMPLOYEE_ssn of Employed = ssn of EMPLOYEE)
THEN *enforcing_action_list.*

WHEN [BEFORE] e2-c2 : salary OF manager MODIFIED
FIRE r2-c2 : IF NOT (salary of EMPLOYEE < salary of manager AND
DEPARTMENT_name of manager = DEPARTMENT_name of employer AND
EMPLOYEE_ssn of Employed = ssn of EMPLOYEE)
THEN *enforcing_action_list.*

WHEN [BEFORE] e3-c2 : EMPLOYEE INSERTED
FIRE r3-c2 : IF NOT (salary of EMPLOYEE < salary of manager AND
DEPARTMENT_name of manager = DEPARTMENT_name of employer AND
EMPLOYEE_ssn of Employed = ssn of EMPLOYEE)
THEN *enforcing_action_list.*

WHEN [BEFORE] e4-c2 : Manages INSERTED
FIRE r4-c2 : IF NOT (salary of EMPLOYEE < salary of manager AND
DEPARTMENT_name of manager = DEPARTMENT_name of employer AND
EMPLOYEE_ssn of Employed = ssn of EMPLOYEE)
THEN *enforcing_action_list.*

WHEN [BEFORE] e5-c2 : DEPARTMENT_name OF Manages MODIFIED
FIRE r5-c2 : IF NOT (salary of EMPLOYEE < salary of manager AND
DEPARTMENT_name of manager = DEPARTMENT_name of employer AND
EMPLOYEE_ssn of Employed = ssn of EMPLOYEE)
THEN *enforcing_action_list.*

WHEN [BEFORE] e6-c2 : DEPARTMENT_name OF Employed MODIFIED
FIRE r6-c2 : IF NOT (salary of EMPLOYEE < salary of manager AND
DEPARTMENT_name of manager = DEPARTMENT_name of employer AND
EMPLOYEE_ssn of Employed = ssn of EMPLOYEE)
THEN *enforcing_action_list.*

WHEN [BEFORE] e7-c2 : Employed INSERTED
FIRE r7-c2 : IF NOT (salary of EMPLOYEE < salary of manager AND
DEPARTMENT_name of manager = DEPARTMENT_name of employer AND
EMPLOYEE_ssn of Employed = ~ n of EMPLOYEE)
THEN *enforcing_action_list.*

WHEN [BEFORE] e8-c2 : EMPLOYEE_ssn OF Employed MODIFIED
FIRE r8-c2 : IF NOT (salary of EMPLOYEE < salary of manager AND

DEPARTMENT_name of manager = DEPARTMENT_name of employer AND
EMPLOYEE_ssn of Employed = ssn of EMPLOYEE)
THEN *enforcing_action_list.*

WHEN [BEFORE] e9-c2 : ssn OF EMPLOYEE MODIFIED
FIRE r9-c2 : IF NOT (salary of EMPLOYEE < salary of manager AND
DEPARTMENT_name of manager = DEPARTMENT_name of employer AND
EMPLOYEE_ssn of Employed = ssn of EMPLOYEE)
THEN *enforcing_action_list.*

The latter illustrates an interesting aspect of the model that was addressed in section 2.4. Recall that "manager" (respectively "employer") is the role of "EMPLOYEE" (respectively "DEPARTMENT") in the relationship "Manages" (respectively "Employed"), and that "salary" is an attribute of "EMPLOYEE" that is inherited by "Manages" (respectively "Employed") and thus by "manager" (respectively "employer"). In addition, "Manages" (respectively "Employed") is usually implemented in a relational database as a column "Manages_DEPARTMENT_name" (respectively "Employed_DEPARTMENT_name") in the relation "EMPLOYEE", i.e., the semantic links provided by "Manages" and "Employed" in the ER model are hidden in the relational model in the form of foreign keys. This information is stored in the meta-database of the schema design and is used for optimizing the derivation of behavior templates (e.g. avoiding redundancy of events like "salary of EMPLOYEE MODIFIED" and "salary of Manages MODIFIED"). It is also used to translate the behavior specification into the DBMS triggers or event alerters (e.g. mapping "Employed INSERTED" to insertion into "EMPLOYEE" relation).

The user (database designer) needs to edit the templates to specify the "BEFORE" clause of the event, usually necessary with rejection actions, and the *enforcing_action_list* for each active behavior. The generation of many templates of active behavior for a single constraint provides modularity and flexibility for the database designer to specify different actions for different constraint-violating events. For example, it might be the user intention to specify the action "REJECT_OPERATION" for rule "r1-c1" and the action "MODIFY EMPLOYEE SET (salary = 10,000)" for rule "r2-c1", although they are used to enforce the same constraint.

3.2 Invariant Properties of the Model

The ER model has a particular set of static constraints, either inherent to the model or implicit in the schema definition, that are implied by the invariant properties of the model.

The enforcement of these constraints can be specified as meta-behavior, i.e., behavior over the meta-database, and automatically generated by the schema design and translation tool for each instance of object types or attributes in the meta-database that is affected by the constraint-violating events. By doing this, the tool relieves the database designer of having to specify individual active database behavior for each affected object or attribute. Of course, this specification must be regenerated every time the meta-database changes, i.e., every time the database schema evolves.

In what follows, we introduce an extension to the notation used so far that allows us to express specification of meta-behavior in a compact form. We use the pseudo-expression "For each" to denote iteration of the specification through the sets of entity and relationship types stored in the meta-database. Hence the following meta-events are defined over the ER objects ENTITY and RELATIONSHIP; each meta-event maps into a set of events in a given actual database schema:

ev1: *ent_name* OF ENTITY INSERTED

ev2: *ent_name* OF ENTITY MODIFIED

ev3: *ent_name* OF ENTITY DELETED

ev4: *rel_name* OF RELATIONSHIP INSERTED

ev5: *rel_name* OF RELATIONSHIP MODIFIED

ev6: *rel_name* OF RELATIONSHIP DELETED

Also, the following meta-schemas of ENTITY and RELATIONSHIP in figure 2.1 are assumed.

ENTITY :

ent_name : name of the entity type.

key_attr_name() : key attribute of the entity type. The notation () means that "key_attr_name" can be composite.

part_rel*(rel_name, part_type) : set of relationship types which the entity type participates in. The notation * means that "part_rel" can be multi-valued. Each "part_rel" is composed by "rel_name" (name of the relationship type), and "part_type" of relationship (type of participation of the entity type in the relationship type, that can be either "Total" or "Partial").

from_ent_conn*(ent_name, conn_type) : set of connections from other entity types, composed of "ent_name" and "conn_type", that can be either "Is_A" (specialization) or "ID" (identification dependency).

to_ent_conn*(ent_name, conn_type) : set of connections to other entity types, i.e., the reciprocal of "from_ent_conn".

RELATIONSHIP :

rel_name : name of the relationship type.

rel_obj*(obj_name, key_attr_name()) : set of object types associated by the relationship type. Each rel_obj is composed by "obj_name" (name of the related entity or relationship type), and "key_attr_name()" (key attribute of the entity or relationship type).

part_rel*(rel_name, type) : same as in ENTITY. Recall that we allow relationships to participate in relationships.

The actual implementation of the meta-database may have a different meta-schema; for example, information on connections between object types are usually kept as meta-attributes of the meta-relationship "ER Connection" (figure 2.2). Here we assume they are stored also in the meta-objects ENTITY and RELATIONSHIP, in order to facilitate the description of conditions and actions in the specification of the meta-behaviors that enforce the invariant properties.

In addition, we use a notation similar to that used in the description of the semantics of the actions (section 2.2.2) to describe the protocols of the meta-behaviors. "Pre-conditions" and "Post-conditions" are defined to apply the rejection strategy, and an "Implies" expression is introduced to apply the propagation strategy.

1. Key constraint (each instance of an entity must be unique). This mapping is given for the sake of completeness: the key constraint is supported declaratively and automatically by most of the DBMSs.

   a. *INSERT* e_i *INTO* E_i
   Pre-condition : $\neg\exists\ e_i \in E_i$

   For each ent_name in ENTITY :
   WHEN BEFORE ev1 : ent_name OF ENTITY INSERTED
   FIRE r1-kc : IF NEW key_attr_name IN SET_OF (key_attr_name)
   THEN REJECT_OPERATION,
   MESSAGE : "Key attribute already exists".

   b. *MODIFY* e_i *IN* E_i
   Post-condition : new key_attribute of e_i = old key_attribute of e_i

   For each ent_name in ENTITY :
   WHEN BEFORE ev2 : ent_name of ENTITY MODIFIED
   FIRE r2-kc : IF NEW key_attr_name $\neq$ OLD key_attr_name
   THEN REJECT_OPERATION,
   MESSAGE : "Not allowed to change key attribute".

2. Relationship referential integrity constraint (a relationship can exist only if the corresponding related objects exist).

a. *INSERT r INTO R where* $R(E_1, ..., E_n)$
Pre-condition : $\exists\, e_1 \in E_1,\ e_1 = E_1(r) \wedge$
$... \wedge$
$\exists\, e_n \in E_n,\ e_n = E_n(r)$

For each rel_name in RELATIONSHIP :
WHEN BEFORE ev4 : rel_name OF RELATIONSHIP INSERTED
FIRE r1-ric : IF NEW rel_obj.key_attr_name NOT IN SET_OF
(rel_obj.key_attr_name)
THEN REJECT_OPERATION,
MESSAGE : "Inexistent related entity".

b. *DELETE* e_i *FROM* E_i
IMPLIES $\forall\, r \in R(e_i)$ *DELETE r FROM R*

For each ent_name in ENTITY :
WHEN ev3 : ent_name OF ENTITY DELETED
FIRE r2-ric : THEN DELETE_RELATIONSHIP part_rel.rel_name
(rel_obj.key_attr_name = OLD key_attr_name) (ev6).

c. *DELETE* r_i *FROM* R_i
IMPLIES $\forall\, r \in R(r_i)$ *DELETE r FROM R*

For each rel_name in RELATIONSHIP :
WHEN ev6 : rel_name OF RELATIONSHIP DELETED
FIRE r3-ric : THEN DELETE_RELATIONSHIP part_rel.rel_name
(rel_obj.key_attr_name = OLD rel_obj.key_attr_name) (ev6).

Note: The latter behavior is necessary because we allow relationship involving relationships. Notice that key_attr_name may be composite; also, notice that part_rel and rel_obj are multi-valued. As a consequence, the mapping from meta-behavior to actual behavior will require multiple iterations through the sets of attributes and objects for the complete specification of conditions and actions.

3. Total participation constraint (if the relationship is total on an entity, then the existence of the related entity requires the existence of this relationship).

a. *INSERT* e_i *INTO* E_i
Pre-condition : $\exists\, R$, *participation of* E_i *in R is total*
Post-condition : $\forall\, R$, *participation of* E_i *in R is total*, $e_i = E_i(r)$

For each ent_name in ENTITY :
WHEN ev1 : ent_name OF ENTITY INSERTED

FIRE r1-tpc : IF part_rel.part_type = "Total"
THEN MESSAGE :
"Need to insert mandatory relationship for inserted entity".

4. Identification dependency constraint (if a relationship is weak on an entity, then the existence of the weak entity requires the existence of the related strong entity).

   a. $DELETE\ e_i\ FROM\ E_i$
   $IMPLIES\ \forall\ e_j \in E_j,\ e_j\ is\ ID\ on\ e_i,\ DELETE\ e_j\ FROM\ E_j$

   For each ent_name in ENTITY :
   WHEN ev3 : ent_name OF ENTITY DELETED
   FIRE r1-idc : IF from_ent_conn.conn_type = "ID"
   THEN DELETE_ENTITY from_ent_conn.ent_name
   (key_attr_name OF from_ent_conn.ent_name =
   OLD key_attr_name) (ev3).

   b. $INSERT\ e_i\ INTO\ E_i$
   Pre-condition : $e_i\ is\ ID\ on\ e_j \rightarrow \exists\ e_j \in E_j$

   For each ent_name in ENTITY :
   WHEN ev1 : ent_name of ENTITY INSERTED
   FIRE r2-idc : IF to_ent_conn.conn_type = "ID" AND
   key_attr_name OF to_ent_conn.ent_name NOT IN SET_OF
   (key_attr_name OF ent_name)
   THEN MESSAGE :
   "Identification dependency: owner entity does not exist".

5. Aggregation referential integrity constraint (an aggregation can exist only if the corresponding components exist).

   This property is enforced by the relationship referential integrity constraint (property 2), since we consider a relationship as an aggregation of the participant entities to allow relationships involving relationships (full aggregation).

6. Superclass completeness constraint (a specialization is total on a generalization, i.e. the existence of a specialization requires the existence of its generalization).

   a. $INSERT\ e_i\ INTO\ E_i$
   Pre-condition : $E_i\ is\ a\ sub-class\ of\ E_j \rightarrow \exists\ e_i \in E_j$

   For each ent_name in ENTITY :
   WHEN ev1 : ent_name OF ENTITY INSERTED

FIRE r1-scc : IF to_ent_conn.conn_type = "Is_A" AND
key_attr_name OF to_ent_conn.ent_name NOT IN SET_OF
(key_attr_name OF ent_name)
THEN MESSAGE :
"Specialization: generic entity does not exist".

b. $DELETE\ e_i\ FROM\ E_i$
$IMPLIES \forall\ E_j\ sub-class\ of\ E_i,\ DELETE\ e_i\ FROM\ E_j$

For each ent_name in ENTITY :
WHEN ev3 : ent_name of ENTITY DELETED
FIRE r2-scc : IF from_ent_conn.conn_type = "Is_A"
THEN DELETE_ENTITY from_ent_conn.ent_name
(key_attr_name OF from_ent_conn.ent_name =
OLD key_attr_name) (ev3).

In the above, we applied both the rejection and the update strategies for deriving meta-behaviors to enforce inherent and implicit constraints of the model. The translation tool can be tailored to support the generation of other active database behaviors that enforce more specific invariant properties like superclass completeness, subclass disjointness, or relationship cardinality constraints. In addition, a more complex constraint-checking behavior can be generated using aggregate predicates, such as: "When a relationship is deleted, if some entity type participation is total on it and the deleted relationship is the last one for the related entity, then send a message to the user, or, alternatively, propagate the deletion to the related entity." All these meta-behaviors embedded in the translation tool can be left as optional so that the user decides at design time which set of invariant properties is desirable to be considered for a given application and which strategy to apply in order to enforce them. Figure 3.2 illustrates the meta-behaviors specified in this section as a meta-$(ER)^2$-diagram of the ER model and its invariant properties.

3.3 Dynamic Constraints

The inherent and implicit constraints implied by the invariant properties of the data model, as well as the explicit constraints exemplified in section 3.1, are constraints that must hold in every state of the database. They are called static constraints because they deal with the consistency of a single database state.

Some explicit constraints deal with the consistency of transitions of database states and are called dynamic constraints. Their specification usually requires very high level predicates that are not expressible in declarative constraint languages. Since they occur less frequently than static constraints, it is easier to specify dynamic constraints procedurally as active database behaviors, rather than augmenting the constraint specification language to capture multiple database states.

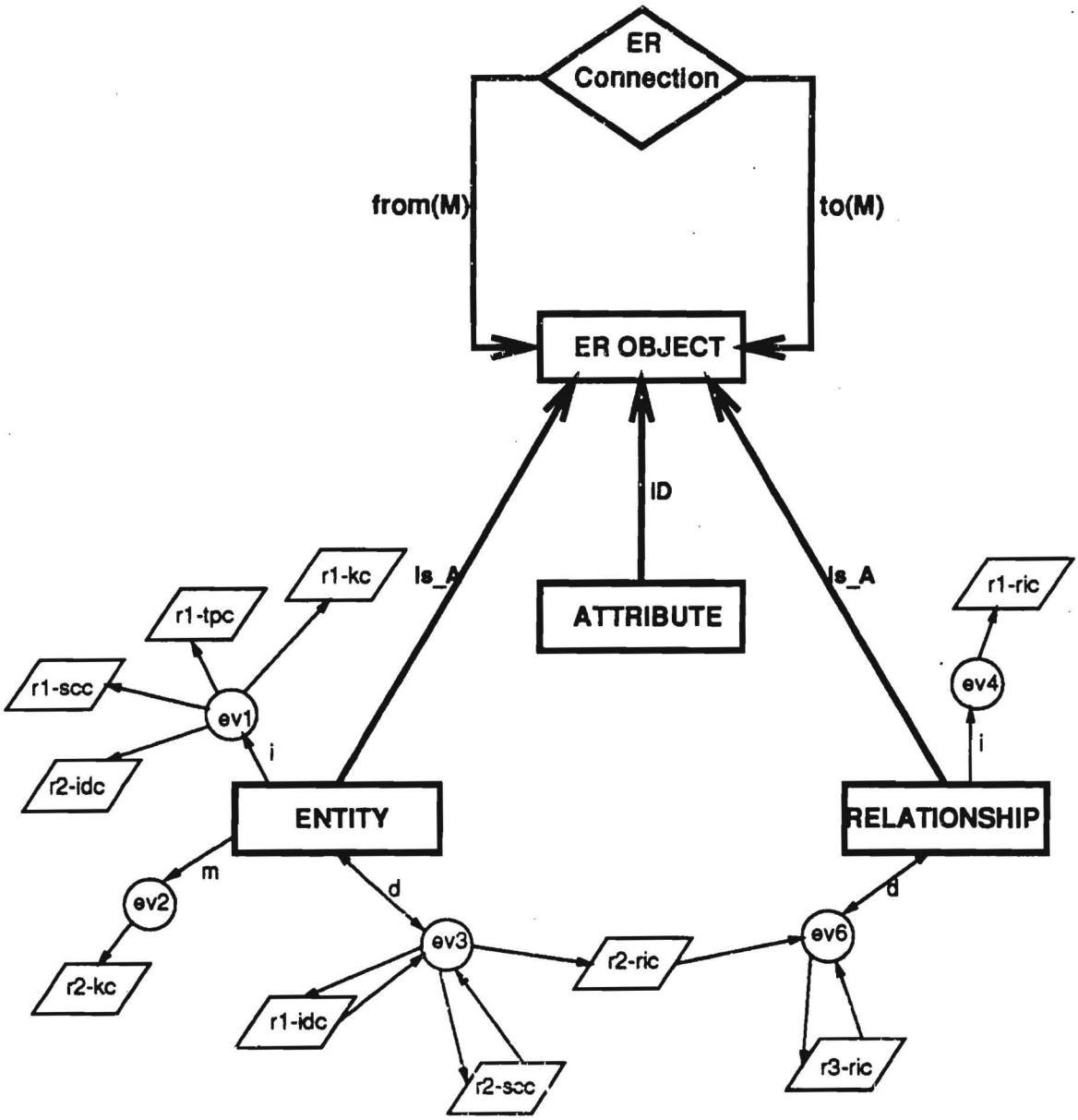

Figure 3.2: Specification of the Invariant Properties as Meta-behaviors

As an example, "the salary of an engineer cannot decrease" is a constraint that requires checking the database states before and after an update on "salary of EMPLOYEE" is performed. Instead of trying to specify such constraint in a rich constraint specification language and then translating it into the procedural constructs that will enforce it, the following active database behavior is easily derived from the semantics of the constraint:

WHEN e1-dc : salary OF EMPLOYEE MODIFIED
FIRE r1-dc : IF OLD job = "engineer" AND NEW salary < OLD salary
THEN MESSAGE : "Engineers' salaries cannot decrease",
REJECT_OPERATION.

3.4 Summary

As a summary, in this section we have shown the application of the active database behavior to enforce integrity constraints. The following conclusions are derived :

1. Static constraints

   (a) If inherent to the data model or implicit in the conceptual schema, i.e., implied by the invariant properties of the model, they can be specified as meta-behaviors. The schema translation tool will instantiate each meta-behavior as actual behavior in the given database and translate them into the language constructs (rules or triggers) in the target DBMS.

   (b) If explicit constraints, i.e., constraints of the application semantics (business rules), they can derive active database behavior by means of an interactive tool that generates templates of behavior specification and then accepts user intervention to complete the specification. For very complex application constraints, the constraint definition may require very complicated predicates, and the mapping process may not be worthwhile, because the constraint-enforcing active behavior must be specified anyway.

2. Dynamic constraints
   These are always explicit, application-oriented constraints. They are better specified directly in the form of the active behavior that will enforce them.

Section 4

CONCLUSION AND FUTURE DIRECTION

Our claim is that the lack of modeling constructs for active database capabilities present in the new generation of relational DBMSs has made it difficult to take full advantage of their potential benefits. The current database design methodology forces the user to defer critical modeling decisions concerning the active behavior of the database to late stages of the design process, where the semantics of the real-world situations are obscured by the intricacies of the implementation model. Because of the inherent complexity of rule-based programming, database designers do not exploit adequately the functionalities of rules, triggers and stored procedures. Furthermore, it is expected that more powerful active capabilities will be added to the DBMSs by demand of non-conventional database applications, enlarging the gap between modeling and specification of executable definitions of active behavior.

Our approach to this problem was to extend the well-established methodology based on the ER model by incorporating active database behavior in the form of events and rules as first-class objects of the model.

The following benefits will result from the extended modeling and design methodology: reduced database design and application development effort with the automatic generation of meta-behavior and translation of active behavior into executable DBMS language constructs; better control of the development of database applications; and better quality of the overall design. In the present report, we have concentrated on the modeling of the active behavior and a specification of the constraints. Further details of the design methodology for active databases can be found in [Tan92].

4.1 Summary

We introduced the $(ER)^2$ model as a uniform way to express active database behavior along with entities and relationships. We separated the concepts of an event and the action that causes its occurrence, many times considered as the same fact in other approaches. We also differentiated events and conditions, because although they both represent predicates and are so modeled in some dynamic modeling approaches, they have different semantics and timing of occurrence and evaluation. With these distinctions, we were able to define both events and rules as objects of the model, rather than only rules as considered in the literature so

far. We identified attributes of events and rules and characterized classes and instances of events and rules. We also identified the inter-event and inter-rule connections, as well as the semantic connections between events and rules, and between events and data objects or the external environment. Based on these modeling concepts and the ER formalism, a set-oriented syntax and semantics for active database behavior was defined. We proposed a diagrammatic representation of active behavior in terms of events and rules, as an extension to ER diagrams. We showed that with the provision of a meta-database of the design, the translation of active database behavior from the $(ER)^2$ model to commercial relational DBMSs can be easily incorporated into the database design process, relieving the user from the need to program rules, triggers and procedures for enforcing that behavior.

Next we showed that constraint maintenance can be achieved by specifying constraints declaratively and deriving appropriate event-condition-action behavior that in effect implement those constructs. This transformation is useful for the types of constraints that, although enforced procedurally by the DBMS, are easier to specify declaratively. From these, one can derive a set of procedures and rules to enforce them. Dynamic constraints, which refer to the consistency of state transitions rather than to a single state, were shown to be more easily specified directly in terms of an active behavior instead of trying to extend the constraint language to consider multiple states. A special type of constraint, which is implied by the invariant properties of the ER model, if not supported declaratively by the DBMS, can be specified by means of a meta-behavior, i.e., behavior over all entity sets and all relationship sets and instantiated to appropriate instances of actual active behaviors by the design and translation tool for a particular populated database.

4.2 Further Research and Development

Supplementary research and development is needed to take full advantage of the benefits that accrue to databases by the incorporation of active capabilities. Some major research directions are listed below:

- It is necessary to combine data, control, and process modeling to capture active database behavior and application transactions in the same model.
- As a consequence, the interaction of rule processing and transaction processing in the execution model of an active database must be considered to provide the database designer with a complete analysis model, in which the whole behavior of the database can be validated.
- A declarative constraint specification language using constraints as predicates at the conceptual level, for enforcing constraints as active behaviors.
- An architecture of tools has been proposed to incorporate the active database extension into current relational database methodology [NTC93]. These tools need to be implemented.

- A graphical interface for specification of events and rules, either integrated with an ER diagramming tool or in a separate editor is needed. Also, a validation tool based on a high level Petri net editor/simulator, possibly taking advantage of the analysis methods developed for hierarchical high-level Petri nets [Jen91] would be desirable.

Furthermore, the research on active databases is raising new issues and discovering new applications; some of them will also impact modeling and design :

- Deductive databases as a class of active databases: a deductive rule can be seen as an active behavior, where there is no event (or it is just a retrieval), and the condition-action pair is a deduction rather than an operation on the database or a message. Since the active database paradigm subsumes the deductive database paradigm, both could be present in actual DBMSs, providing a platform for large knowledge bases and expert systems [SKdM92].

- Rules in OO DBMSs: the OO paradigm seems to be a natural way to accommodate active behavior in the form of events and rules as first-class objects. The availability of efficient implementations of rules in OO DBMSs is expected and will impact the way active database behavior is modeled and designed [DPG91].

- Parallel and distributed active databases: rule processing is usually performed in a centralized, sequential fashion. Given the high interest in parallel and distributed environments, it is important for active databases to be adapted to them [CW92].

- Database authorization schema: the active database paradigm is clearly a real alternative for database security, and much work has to be done in this area [Lun92].

- Derived data maintenance: it is widely recognized that the active database paradigm can be used to automatically maintain derived data such as views. Research on design and analysis of active behavior for efficiently maintaining derived data is on-going [CW91].

- Schema evolution: automatic propagation of changes in the schema can be performed using the active database paradigm, especially by taking advantage of the meta-database that describes the database and its design process [MR90].

- Reverse engineering of legacy systems: in spite of the wide acceptance of relational database technology, most of the corporate data is currently stored in large data repositories residing in flat files. Reverse engineering of these old systems is a key research area, and the active database paradigm can play an important role in the knowledge discovery of business rules.

Bibliography

[AH85] S. Abiteboul and R. Hull. Update propagation in the IFO database model. In *Proceedings of the International Conference on Foundations of Data Organization*, 1985.

[CERE88] B. Czejdo, R. Elmasri, M. Rusinkiewcz, and D.W. Embley. Semantics of update operations for an extended entity-relationship model. In *Proceedings of the ACM Annual Computer Science Conference*, 1988.

[Cha91] S. Chakravarthy. Active database management systems: requirements, state-of-the-art, and an evaluation. In H. Kangassalo, editor, *Proceedings of the International Conference on the Entity Relationship Approach*, 1991.

[CW90] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proceedings of the International Conference on Very Large Data Bases*, 1990.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the International Conference on Very Large Data Bases*, 1991.

[CW92] S. Ceri and J. Widom. Production rules in parallel and distributed database environments. In *Proceedings of the International Conference on Very Large Data Bases*, 1992.

[DEC89] DEC. *Vax RDB/VMS SQL Reference Manual*, 1989.

[DPG91] O. Diaz, N. Paton, and P. Gray. Rule management in object oriented databases: a uniform approach. In *Proceedings of the International Conference on Very Large Data Bases*, 1991.

[GJS92] N.H. Gehani, H.V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model and implementation. In *Proceedings of the International Conference on Very Large Data Bases*, 1992.

[ING90] INGRES. *INGRES/SQL - Reference Manual for the Unix and VMS Operating Systems*, 1990.

[INT90] INTERBASE Software Corporation. *InterBase DDL Reference Manual - Interbase version 3.0*, 1990.

[Jen91] K. Jensen, editor. *High-Level Petri Nets: Theory and Applications.* Springer-Verlag, 1991.

[Kos92] E. Kosciuszko. How to implement integrity in Oracle. *Database Programming and Design*, 5(7), July 1992.

[Lun92] T.F. Lunt. Security in database systems: a research perspective. *Computers and Security*, 11(1), 1992.

[Mel90] J. Melton. *ISO/ANSI Working Draft Database Language SQL2.* ISO/ANSI, 1990.

[MF91] V.M. Markowitz and W. Fang. SDT: A database schema design and translation tool - reference manual. Technical Report LBL-27843, Lawrence Berkeley Laboratory, 1991.

[Mis91] D. Mishra. SNOOP: An event specification language for active databases. Master's thesis, University of Florida, 1991.

[MLM+92] V.M. Markowitz, S. Lewis, J. McCarthy, F. Olken, and M. Zorn. Data management for genomic mapping applications: a case study. In *Proceedings of the Conference on Scientific and Statistical Database Management*, 1992.

[MR90] L. Mark and N. Roussopoulos. Information interchange between self-describing databases. *Information Systems*, 15(4), 1990.

[NB91] S.B. Navathe and A. Balaraman. A transaction architecture for a general purpose semantic data model. In T.J. Teorey, editor, *Proceedings of the International Conference on the Entity Relationship Approach*, 1991.

[NTC93] S.B. Navathe, A.K. Tanaka, and S. Chakravarthy. Active database modeling and design tools: Issues, approach, and architecture. *IEEE Database Engineering*, 1993. To appear.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[SA85] R. Snodgrass and I. Ahn. A taxonomy of time in databases. In *Proceedings of the International Conference on Management of Data*, 1985.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching, and views in data base systems. In *Proceedings of the International Conference on Management of Data*, 1990.

[SKdM92] E. Simon, J. Kiernan, and C de Mandreville. Implementing high-level active rules on top of relational databases. In *Proceedings of the International Conference on Very Large Data Bases*, 1992.

[SM91] E. Szeto and V.M. Markowitz. ERDRAW: A graphical schema specification tool - reference manual. Technical Report PUB-3084, Lawrence Berkeley Laboratory, 1991.

[SYB87] SYBASE Inc. *Transact-SQL User's Guide*, 1987.

[Tan92] A.K. Tanaka. *On Conceptual Design of Active Databases.* PhD thesis, Georgia Institute of Technology, 1992.

[TNCK90] A.K. Tanaka, S.B. Navathe, S. Chakravarthy, and K. Karlapalem. Toward conceptual modeling of active databases. Technical Report TR-90-21, University of Florida, 1990.

[TNCK91] A.K. Tanaka, S.B. Navathe, S. Chakravarthy, and K. Karlapalem. ER-R: an enhanced ER model with situation-action rules to capture application semantics. In T.J. Teorey, editor, *Proceedings of the International Conference on the Entity Relationship Approach*, 1991.

[WF90] J. Widom and S.J. Finkelstein. Set-oriented production rules in relational database systems. In *Proceedings of the International Conference on Management of Data*, 1990.