ENTITY-RELATIONSHIP APPROACH
S.T. March (Editor)
Elsevier Science Publishers B.V. (North-Holland)
© ERI, 1988

THE CHRIS CONSULTANT

Antonio L. Furtado*, Marco A. Casanova, Luiz Tucherman

IBM Brasil - Rio Scientific Center
P.O. Box 1830
20.071, Rio de Janeiro, Brasil

CHRIS is an expert software tool to help in the design of information systems containing a database component. CHRIS involves an extended entity-relationship information model, the relational data model and a database management system. A prototype version of the tool, written in Prolog extended with a query-the-user facility, is fully operational. The prototype includes an interface for experimental use which enforces the integrity constraints of the application.

1. INTRODUCTION

CHRIS is an expert software tool to help in the design of information systems containing a database component. As an acronym, CHRIS stands for "Concepts and Hints for Relational Interfaces Synthesis" and it is also intended as a homage to Chris Date. In fact, the method adopted is close to what he calls 'the synthetic approach' [13].

The tool follows a design strategy consisting of three phases:

1. *the conceptual design phase*, where the designer specifies the application in terms of an extended entity-relationship model. The designer is also prompted to provide additional information about how to enforce the integrity constraints.
2. *the logical design phase*, where the ER schema is automatically mapped into a normalized relational schema;
3. *the interface generation phase*, that synthesizes DDL commands to create the data base plus an interface for experimental use, thus providing the benefits of running specifications. The interface enforces the integrity constraints defined in the first phase.

The major contribution of the work is then a tool covering the complete design cycle, that is, mapping a high-level schema specification into an interface that transforms operations into correct transactions. The tool therefore compensates the lack in data models and, to an even worse degree, in database management systems of a more comprehensive set of constructs or commands for integrity preservation.

The tool is written in VM/Prolog [26], with the addition of the ETC extension to VM/Prolog [17], whose purpose is to help in the construction of expert systems in

---

* On leave from the Pontificia Universidade Catolica do Rio de Janeiro

general, and which has been largely influenced by APES [18]. The major function provided by ETC is dialogue-management. SQL/DS [22] is used, across the interface between this system and VM/Prolog.

The idea of transforming an extended entity-relationship schema into a relational schema is not new [2,3,4,12]. The difference between most of the proposals lies in the way the structures of the ER Model are translated into structures of the Relational Model based on the scope of the extensions adopted. Some of the methods proposed are performance-driven [5].

Most of the research on the ER Model concentrated on the static view of entities and relationships. However, petri-net techniques have been used to take into account the dynamic behavior of the entities and relationships [1,9,23,24]. Our approach captures such dynamic behavior in terms of restrict/propagate rules, which are then used by a specially designed interface to transform an update operation into a correct transaction, following an idea proposed in [25].

Research on expert tools for database design can be found in [6,7,20,21]

The paper is divided as follows. Section 2 reviews some basic concepts of the Entity-Relationship Model and introduces different kinds of integrity constraints. Section 3 discusses how to preserve integrity constraints in the ER Model. Section 4 outlines the design strategy the tool implements and the overall structure of the tool. Section 5 presents a prototype version of the tool.

2. A BRIEF REVIEW OF BASIC CONCEPTS

2.1 The ER Model

We shall use a variation of the original ER Model [10] that incorporates abstraction hierarchies [8,19,25] and certain types of integrity constraints.

An *ER schema* declares a set of *entities*, *relationships* and *attributes* and, for each entity or relationship, indicates its attributes and, for each relationship, the list of participating entities. A relationship is *binary* iff it has exactly two participating entities. If two entities or relationships O and O′ have an attribute with the same name A, then we use A.O and A.O′ to distinguish the two occurrences.

The schema also defines a set of *integrity constraints*, discussed in the next subsections. We shall see that important kinds of integrity constraints commonly associated with the ER model may be reinterpreted as variations of referential integrity [14].

Briefly, referential integrity is a notion originally proposed in the context of relational databases. It means that one may declare that the projection of a table T on a set of columns **c** must always be a subset of the projection of a table T′ on a set of columns **c**′ that form a key of T′. The set **c** constitutes a *foreign key*. Informally, it means that if tuples in T reference tuples in T′, the latter must in fact exist in T′. In other words, referential integrity excludes "dangling" references. Note also that referential integrity is just a special case of inclusion dependencies [11]. But tables

are simply a representation for information objects, and therefore references across tables should reflect some kind of reference perceivable in the information realm.

A *state* **s** of an ER schema associates:

- to each entity E named in the schema an *entity set* **s**(E),
- to each relationship R between entities $E_1,...,E_n$ a *relationship set* **s**(R), which must be a subset of the cartesian product of $\mathbf{s}(E_1) \times ... \times \mathbf{s}(E_n)$, and
- to each attribute A of an entity E (or relationship R), a total or partial *attribute function* **s**(A) whose domain is **s**(E) (or **s**(R)).

Each element *e* of **s**(E) (or **s**(R)) is called an *instance* of the entity E (or relationship R) and the range of the attribute function for an attribute A is called the *attribute value set* of A.

Finally, if A is an attribute of an entity or relationship O and **s** is a state of the schema, we will use A(*e*) instead of **s**(A)(*e*). Similarly, if **K** is a sequence $(K_1,...,K_n)$ of attributes of O, we will use **K**(*e*) instead of $(\mathbf{s}(K_1)(e),...,\mathbf{s}(K_n)(e))$.

2.2. Integrity Constraints over Attributes

An ER schema must declare the *type* of each attribute A, taken from a few standard data types, to provide a convention for the representation on a machine or other medium of the attribute value set of A. Typical data types are integers, floating-point numbers, character strings, etc. For machine representation, all values of an attribute are usually required to come from a single data type, although in the real world there exist attributes with values coming from heterogeneous data types, such as the names of playing cards.

The type of an attribute may imply that the attribute is *single-* or *multi-valued.* For the entity PERSON, for example, attribute NAME is normally declared to be single-valued (a person has only one name), whereas TELEPHONE-NUMBER may well be declared to be multi-valued.

An ER schema may also declare that an attribute A of an entity E *does not admit undefined values.* In this case, a state **s** *satisfies* such constraint iff **s**(A) is total. For example, the attribute DRIVER'S-LICENSE of entity PERSON admits undefined values, but certainly NAME does not.

2.3 Integrity Constraints over Entities

2.3.1 Keys

Each instance of an entity E should have a defined value for at least one attribute that distinguishes that instance from other instances in the same set. Thus, we might refer to an instance of PERSON as 'John' and to another instance whose name we ignore by his driver's licence.

A *key* for an entity E is a sequence **K** of attributes of E, possibly with just one element, such that

K1. each attribute in **K** is single-valued
K2. each attribute in **K** does not admit undefined values

A state **s** *satisfies* **K** or is *consistent* with the declaration of **K** iff, for any two instances e,f in **s**(E), if **K**(e) = **K**(f) then $e = f$.

The requirement that each attribute in **K** be single-valued could be dropped. For example, assume that entity FOOTBALL-TEAM has an attribute COLOURS and that each team is uniquely identified by its colours. Then COLOURS can serve as a multi-valued key.

For practical reasons, we also require that:

K3. an ER schema must declare at least one key for each entity. If more than one key is declared, the schema must select one of them to be the *primary key* for the entity.

2.3.2 is-a Hierarchies

Several kinds of hierarchies of entities have been proposed, among which the best-known are **is-a** and **part-of**. Here, we shall concentrate on the **is-a** hierarchy.

Suppose that an ER schema has two entities E and F. Then, at the schema level, the constraint "E **is-a** F" has the following effects:

ISA1. all attributes of F are inherited by E, that is, automatically become attributes of E;
ISA2. each key of F also becomes a key of E;

Naturally, E can have additional attributes, besides those inherited from F.

A state **s** *satisfies* "E **is-a** F" iff

- **s**(E) $\subset$ **s**(F)
- for each attribute A of F, for each instance e in **s**(E), **s**(A.E)(e) = **s**(A.F)(e).

Therefore, the **is-a** constraint may be classified as a referential integrity constraint.

As an example, consider the entities PERSON and EMPLOYEE and assume that EMPLOYEE **is-a** PERSON. Hence, if PERSON has the attributes NAME and DRIVER'S-LICENCE, EMPLOYEE also has these attributes by inheritance. But EMPLOYEE could also have, new attributes such as SALARY, EMPLOYEE-NUMBER, etc... If NAME is a key of PERSON, then it must also be a key of EMPLOYEE. So, for example, John is John, both as a person and as an employee.

For practical reasons, one may also consider the following extra requirement:

ISA3. the primary key of F is also the primary key of E.

This last requirement is convenient because it helps deciding when an entity instance of E is the same as an entity instance of F. Indeed, if **K** is the primary key of both E and F, e and f are the same instance occurring in E and F, respectively, iff **K**(e) = **K**(f).

2.4 Integrity Constraints over Relationships

If R is declared as a relationship between entities $E_1,...,E_n$ in an ER schema then, by definition, in any state **s** of the schema, if $(e_1,e_2,...,e_n) \in \mathbf{s}(R)$, then $e_i \in \mathbf{s}(E_i)$, for each i=1,...,n. To simplify the discussion let us name this the *incidence constraint* or *I-constraint*, for short ('incidence' is taken from graph theory).

The I-constraint says, therefore, that the existence of a relationship instance always depends on the existence of the participating entity instances. To give an example, an instance of the ASSIGNMENT relationship between EMPLOYEES and PROJECTS can only exist if both the EMPLOYEE and the PROJECT instances exist.

However, a constraint in the opposite direction can also be characterized. In an ER schema, one may declare that a relationship R over $E_1,...,E_n$ is *total* over E_i, for some i in [1,n]. We name this the *totality constraint* or *T-constraint*, for short.

A state **s** of the schema *satisfies* this constraint iff, for each $e_i \in \mathbf{s}(E_i)$, there exists $(f_1,f_2,...,f_n) \in \mathbf{s}(R)$ such that $f_i = e_i$.

An example of a T-constraint arises if, typically as an enterprise policy, an employee cannot exist unless he is assigned to at least one project. In this case we say that relationship ASSIGNMENT is total with respect to EMPLOYEE.

We stress the fact that, while the I-constraint always applies by definition, a T-constraint must be explicitly established for specific entities and relationships in an ER schema. T-constraints reflect policies that an enterprise may adopt.

Note that both the I-constraint and the T-constraint may also be classified as referential integrity constraints.

An ER schema may also declare that a relationship R over entities $E_1,...,E_n$ is *functional* over an entity E_i, $1 \leq i \leq n$., in which case we also say that E_j, with $i \neq j$, *is functionally dependent* on E_i in R. A state **s** of the schema *satisfies* this constraint iff, for any two instances $(e_1,e_2,...,e_n)$ and $(f_1,f_2,...,f_n)$ in $\mathbf{s}(R)$, if $f_i = e_i$ then $f_j = e_j$, for each j in [1,n].

Finally, let R be a binary relationship over E and F. If R is functional and total over F, then we say that F is a *weak entity* depending on E in R.

3. PRESERVING INTEGRITY IN THE ER MODEL

3.1 Statement of the Problem

As discussed in section 2, an ER schema may contain several integrity constraints. A state of the schema is *consistent* iff it satisfies all constraints of the schema and a transaction is *correct* iff it maps consistent states into consistent states.

A fundamental question therefore is how to transform a transaction specified by the user into a correct transaction. This question brings up another issue since some types of integrity constraints do not completely determine the transformation. Thus, in order to solve the basic question, we must also extend the concept of integrity constraint to include extra information that specifies how a transaction must

be modified. This is a very important issue that is often neglected, although the CODASYL DBTG proposal had already made some progress in this direction.

We shall look in this section into a special case of this problem, defined as follows:

- How to transform an insertion or deletion into a correct transaction with respect to the type of constraints defined in section 2.
- What kind of extra information must be included along with the integrity constraints of section 2 to disambiguate the transformation of such simple operations.

Note that to transform updates one may simply combine the transformations associated with insertions and deletions.

3.1 Preserving Integrity Constraints over Attributes

Deletions always preserve integrity constraints over attributes. However, an insertion **i** of an instance *e* of entity E must be rejected, when executed beginning on a state **s**, in the following cases:

A1. **i** supplies values for the attributes of E that do not come from the prescribed data types;
A2. **i** omits the value of an attribute A that does not admit undefined values for E (which is always the case when A belongs to a key);
A3. there exists an instance *f* in **s**(E) and a key **K** of E such that **K**(*e*) = **K**(*f*).

Similar criteria applies to insertions over a relationship.

3.2 Preserving is-a Hierarchies

Suppose that an ER schema declares that E **is-a** F. We must extend such declaration to indicate whether:

- beginning on a state **s**, an insertion **i** of an instance *e* of E must:
  EE1. be rejected if *e* is not already in **s**(F)
  EE2. be followed by an insertion of *e* as an instance of F
- beginning on a state **s**, a deletion **d** of an instance *f* of F must:
  EE3. be rejected if *f* also exists in **s**(E)
  EE4. be followed by a deletion of *f* as an instance of E

Note that the new insertion generated in choice EE2 is completely defined by **i**, since all attributes of F are also attributes of E.

Since **is-a** is transitive, such transformations may propagate through several layers of the **is-a** hierarchy.

3.3 Preserving I-Constraints

Suppose that an ER schema declares that R is a relationship over $E_1,...,E_n$. We must extend such declaration to indicate whether:

- an insertion **i** of an instance $(e_1,...,e_n)$ of R must:
  ERI1. be rejected if no instance e_i of E_i exists, for each i in [1,n]
  ERI2. be followed by an insertion of e_i into E_i, if e_i is not an instance of E_i.
- a deletion **d** of an instance e_i of E_i must:
  ERI3. be rejected if an instance $(f_1,...,f_n)$ of R with $f_i=e_i$ exists
  ERI4. be followed by a deletion of all instances $(f_1,...,f_n)$ of R with $f_i=e_i$

On the other hand, insertions of entities and deletions of relationships cause no problems.

As an example, consider two entities, EMPLOYEE and PROJECT, and a relationship, ASSIGNMENT, between these two entities. Choice ERI1 implies that to insert an assignment of employee John to project Alpha, both John and Alpha must previously exist.

Choice ERI2 may seem unusual in the sense that the insertion in R does not completely determine the insertion in E_i. However, it becomes perfectly reasonable if we recall that such choice will help transform the insertion in R into a transaction that may ***ask*** the user any additional information needed to create the insertion in E_i.

Choice ERI3 must be adopted if, say, there is a policy saying that employees assigned to projects cannot be deleted. Note that the symmetric policy also makes sense, since it may be the case that projects having employees assigned to it cannot be deleted.

However, the enterprise policies may dictate that employees may be deleted, regardless of their participation in projects. In this case, the I-constraint imposes choice ERI4: the assignment of John to any projects ceases when John is deleted. The policy of allowing the deletion of projects even with assigned employees also makes sense, and again results in choice ERI4: the assignment of employees to Alpha is deleted if Alpha is deleted.

No matter what choice is made, deletions of entity instances remain possible. The difference lies only in ordering the operations appropriately. If ERI3 is adopted for some E_i, then all participations of an instance e_i in R must be deleted before e_i is deleted. In the example, before deleting John we should remove him from all his assignments.

3.4 Preserving T-Constraints

Suppose that an ER schema declares that R is a relationship over $E_1,...,E_n$ and that R is total over E_i.

In this case, a single possibility for transforming an insertion **i** of an instance e_i of E_i presents itself:

ERT1. the insertion **i** must be followed by the insertion of an instance $(f_1,...,f_n)$ of R such that $f_i = e_i$.

ERT1 means that, if a correct transaction contains an insertion in E_i, it must also contain an insertion in R. But which instances $f_1,f_2,...,f_{i-1},f_{i+1},...,f_n$ should we take? Unless some criterion has been chosen, we are left with the need to consult the person performing the insertion of e_i. In the example, suppose that the ASSIGNMENT relationship is total with respect to EMPLOYEE. Then, John must at all times be assigned to at least one project, and hence the insertion of John propagates to the insertion of an assignment of John to some existing project. To which project is John to be assigned will depend of the decision of the person in charge.

The situation is in fact more complex because a relationship may be total with respect to more than one entity. For example, suppose that ASSIGNMENT is total with respect to both EMPLOYEE and PROJECT, so that all employees must be assigned to projects and all projects must have employees assigned to them. Suppose that, when we proceed to the insertion of John, no projects have been created so far. Then, a correct transaction must at least contain the insertion of John in EMPLOYEE, the insertion of a new project, say, Alpha in PROJECT and the insertion of John assigned to Alpha in ASSIGNMENT.

A simplifying decision would be to permit a relationship to be total with respect to at most one participating entity. The rationale for this decision is that it seems less natural to propagate the insertion of relationship instances to the insertion of entity instances. In the example, we would be creating a project to justify the insertion of John. Nevertheless, the decision remains an arbitrary one.

Recall that we assumed that R is total with respect to E_i. To disambiguate the treatment of a deletion **d** of an instance $(e_1,...,e_n)$ of R, beginning on a state **s**, we must extend the declaration of T-constraints to indicate whether to:

ERT2. reject **d** if there is no other instance $(f_1,...,f_n)$ in **s**(R) with $f_i = e_i$, or

ERT3. force **d** to be followed by a deletion of e_i from E_i, if there is no other instance $(f_1,...,f_n)$ in **s**(R) with $f_i = e_i$

Choice ERT2 is the correct one, for example, when ASSIGNMENT is total with respect to EMPLOYEE (the case of totality with respect to PROJECT is analogous) and the enterprise policies rule out deletion of an assignment if it is the only one involving a given employee. Choice ERT3 applies instead if the policies indicate that the deletion of the last assignment propagates to the deletion of the employee.

The choices related to T-constraints are not independent from those related to I-constraints.

For example, if both ERI3 and ERT2 hold, a rather peculiar situation arises. Imagine that these two choices are taken in our example. Then, once inserted, an employee can never be deleted as a single operation, since by ERI3 before deleting an employee his assignments should be deleted, but by ERT2 the last assignment cannot be deleted. This is another situation where a transaction must necessarily contain at least two operations.

As another example, we have that choice ERI4 implies choice ERT3, that is, choices ERI4 and ERT2 are incompatible. Indeed, let R be a relationship over entities $E_1,...,E_n$. Suppose that R is total over E_i and that the designer fixes choice ERI4 for

R and E_j, $j \neq i$. Then, a deletion from E_j must be followed by deletions from R, whose execution may require a deletion from E_i. Therefore, choice ERT3 must hold for R and E_i.

Finally, insertions of relationships as well as deletions from entities pose no problems for T-constraints.

4. AN OUTLINE OF THE DESIGN TOOL

This section briefly outlines the design strategy the tool implements and the overall structure of the tool.

The design strategy that the tool follows has three phases:

The Conceptual Design Phase

The tool extracts from the designer a specification of the application in terms of the entity-relationship model, extended to include **is-a** hierarchies and the other constraints, described in section 2. During the specification of the constraints, the tool also asks for additional information, as discussed in section 3, to disambiguate the treatment of operations.

The Logical Design Phase

The tool maps the ER schema obtained in the previous phase into a relational schema.

The Interface Generation Phase

The tool first maps the relational schema into SQL DDL commands and passes them to SQL/DS to create the data base. It then maps the additional information provided with the integrity constraints into *restrict rules*, that indicate when to reject operations, and *propagation rules*, that indicate what extra operations must be triggered. Such rules are expressed in VM/Prolog combined with SQL/DS commands and stored in a permanent file.

The tool also provides an interface for monitored manipulation of the final SQL/DS database that can be used for experimentation and testing. The interface runs under VM/Prolog and accepts SQL/DS insertion, deletion and update commands on the SQL/DS tables. It then handles such commands with the help of VM/Prolog predicates that interpret the restrict and propagate rules generated during the third phase. The net effect is to transform such commands into transactions correct with respect to the constraints inherited from the first design phase. This approach is reminiscent of the initial proposal of System R that included assertions and triggers [15,16].

The tool is now fully operational as a prototype. At the conceptual design phase, an extension to VM/Prolog - called ETC - is used to provide a controlled query-the-user capability. At the logical design phase VM/Prolog is used exclusively. At the interface generation phase, VM/Prolog is combined with SQL/DS.

The initial version of the tool was somewhat limited in order to be available in a reasonably short time. Yet, even with the limits imposed, the implementation turned out to be quite complex.

Some of the limits imposed affect the design strategy itself, the most important being:

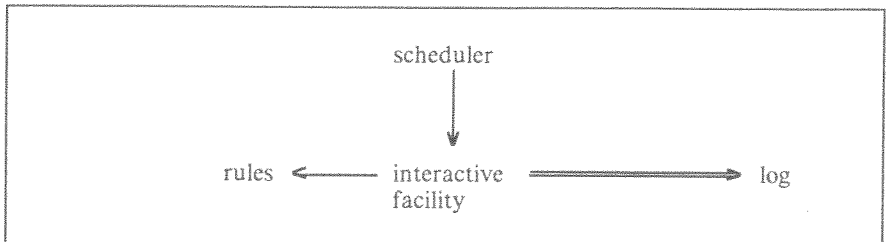
- Attribute types are simply those of the underlying DBMS.
- Each entity can have only one key, which may be simple or composite. At the relational level, this means that alternative 'candidate keys' are not considered.
- Each entity F can occur on the left-hand side of at most one **is-a** constraint. That is, the set of **is-a** constraints induces a strict hierarchy of entities (and not a partial order).
- An entity can participate only once in a relationship, that is, there is no provision for participation of entities in two or more **roles**. A familiar example where the roles concept arises would be that of a REPORTS-TO relationship, with entity EMPLOYEE participating twice (in the roles of SUPERIOR and SUBORDINATE).
- A relationship can be declared total with respect to no more than one participating entity.
- Only binary relationships can be declared to be one-to-n.
- At the conceptual design phase the designer can choose between restriction or propagation only in certain cases of I-constraints and T-constraints.
- Non 3rd NF tables may be generated in one specific case.

The implementation was likewise simplified. Only at the conceptual design phase the prototype establishes a dialogue with the user. The logical design and interface generation phases are totally automatic. When running the interface, interaction with the user consists of error messages, simple traces of invoked and triggered SQL/DS commands and questions to the user in cases where he must supply values to propagate insertions and updates.

5. A PROTOTYPE VERSION OF THE TOOL

5.1. Implementation of the Conceptual Design Phase

This phase is best characterized as a knowledge-acquisition phase, during which the general knowledge about the design strategy embedded in the tool is complemented by knowledge about the specific application being designed. The tool must acquire this specific knowledge by establishing a rule-driven dialogue with the designer. Once logged, the answers supplied by the designer become a fundamental part of the design. The ETC extension to VM/Prolog [17] provides a dialogue capability following this framework, illustrated in the figure below. The single arrows denote program invocation, and the double arrow denotes output.



The design rules that capture the general knowledge about the design strategy are expressed, in a suitably declarative style, mostly by certain special classes of ETC predicates. In all of them, the *first argument*, X, is the pattern of the query being posed to the user, typically of the form p(<parameter-list>). A *second argument*, Y, may be present, whose meaning will be explained for each kind of predicate.

We first consider the two kinds of predicates related to correctness:

1. valid_answer(X,Y) - Y is a VM/Prolog expression that must be satisfied with the variables in X replaced by the values supplied by the user's reply;
2. trigger(X,Y) - Y is a VM/Prolog expression containing executable commands, usually to add or delete clauses to/from the workspace.

The kinds of predicates related to completeness are:

1. unique_answer(X) - simply informs that only one valid reply is to be expected, so that the user will not continue to be prompted for further replies to the query X;
2. complete_answer(X, <Y,M>) - Y is a VM/Prolog expression that is evaluated, using the values supplied by the user, as soon as he types "end"; if Y is not satisfied, X is asked again; M is a message (but can be a null character string) to warn the user about the insufficiency of his reply.

Finally, some predicates merely provide natural-language-like formulations of queries to the designer and of listings of logged predicates (for details see [17]).

We are now in a position to give a schematic view of the two schedulers and of the various correctness and completeness predicates attached to queries, which govern the conceptual design phase. The design of entities is treated separately from the design of relationships.

***When specifying an entity E*:**

Scheduler:

If other entities have been defined before, is there some entity F such that E **is-a** F?

For each attribute A of E not inherited from F,

What is the type of A?

Is A multi-valued? if not, can its value be undefined?

Which are the attributes of the key of E? (to meet requirement K3 and the additional restriction that there is only one key)

valid_answer:

If E **is-a** F, then F must be a pre-defined entity

The type of an attribute must be one of char(N) (where N is an integer), integer or float

Each attribute in the key cannot be undefined or multi-valued (requirements K1 and K2)

trigger:

If E **is-a** F, then E inherits all attributes of F plus their properties of being or not multi-valued and accepting or not an undefined value; the keys of E are exactly those of F (requirements ISA1 and ISA2)

If an attribute is multi-valued, it may also have an undefined value

unique_answer:

The type of an attribute is unique

complete_answer:

E must have at least one identifying attribute, i.e. the key must be defined (requirement K3)

When specifying a relationship R:

Scheduler:

Which are the participants of R;

For each participant E_i,

Is R functional with respect to E_i?

If no other participant E_j has been so declared, is R total with respect to E_i? If yes, then must an instance of E_i be deleted when its last participation in R is deleted? (i.e. which of the choices - ERT2 or ERT3 - applies?)

Can an instance of E_i participating in R be deleted? (i.e. which of the choices - ERI3 or ERI4 - applies?)

For each attribute A of R,

What is the type of A?

Is A multi-valued? if not, can its value be undefined?

Valid_answer:

Participants must be previously defined entities

Trigger:

If E_i participates in R, then the keys of E_i are part of those of R and are (for convenience) directly included among the attributes of R

If an attribute is multi-valued, it may also have an undefined value

Complete_answer:

A relationship must have at least two participants

5.2. Implementation of the Logical Design Phase

Four types of tables are considered when mapping from ER schemas into relational schemas:

1. entity tables;
2. relationship tables;
3. extended-entity tables;
4. repeating-attribute tables.

Entity tables are named after the entity that they represent. The columns of a table **E** (representing entity E) are the single-valued attributes of E. From the attributes inherited from an F such that E **is-a** F only those that belong to the key of F are included.

Relationship tables are named after the relationships that they represent. If **R** is a table representing R, its columns are the keys of the participating entities plus the

single-valued relationship attributes. However relationship tables are not created for relationships that are binary and functional with respect to one of the two participating entities.

If R is a binary relationship over E_i and E_j and R is functional with respect to E_i, then no relationship table will be created for R. The table for E_i, instead of being an ordinary entity table, will be an *extended-entity table* $\mathbf{E}_i$ that has columns representing the attributes of E_i, the attributes on R and those in the key of E_j. The use of extended-entity tables reduces the number of tables and, as a consequence, may decrease the number of accesses. Note that, if E_i participates in other relationships with the above properties, their attributes will also result in columns of the same table $\mathbf{E}_i$.

Finally, *repeating-attribute tables* exist as 'extensions' to other tables to represent multi-valued attributes of entities and relationships. Their names are formed by a concatenation of the entity (or relationship) name with the name of one multi-valued attribute, for example, **E_A**, for an entity E having a multi-valued attribute A. In this example there will be columns for the keys of E and one column for A. The reason for these tables is to comply to the first-normal-form requirement of relational databases [14].

We now indicate the criteria for determining which columns constitute the key for each kind of table. For entity tables, the key is formed by the attributes of the entity key. For relationship tables, the key is formed by the sequence of the keys of the participating entities. For extended-entity tables, the key contains only the attributes of the entity key. This is a useful consequence of the relationship being functional with respect to the entity, and it is the reason why this compact table organization is possible. Repeating-attribute tables are 'all key tables', i.e. the key comprises all columns.

In the present version of the tool, the operational behaviour that is required to enforce referential integrity has been directly treated in the interface generation phase, even though it seems useful to also make it explicit at the logical design phase. For consistency with the actual implemented tool, discussion of this topic will be deferred to the next section.

5.3. Implementation of the Interface Generation Phase

This phase begins with the creation of the tables determined in the logical design phase, via the SQL/DS 'create table' command. Columns are associated with the data type indicated in the conceptual design phase. Also, the following measures are taken for the sake of preserving integrity:

- columns are defined with the declared data types;
- columns whose values cannot be left undefined are declared with the 'not null' property;
- indices with the 'unique' property are created (via 'create index') to ensure the uniqueness of simple or compound keys;

Note that these three general decisions pass on to the DBMS the task of checking update requirements A1, A2 and A3.

Then, a VM/Prolog interface for monitored database manipulation is created. The interface combines a specific *application component*, automatically generated by the tool, with a *general component*, which has been hand-programmed once and for all and should, in principle, be able to handle a large class of applications.

The most important element of the general component is the 'execute' predicate, which provides monitored execution of data manipulation operations. Let O be an insert, delete or update operation. Before performing O, 'execute' retrieves information from the database that will be used to process restrict and propagate rules. If O is a delete operation, the set of tuples to be deleted is retrieved. If O is an update operation, a set of pairs <old tuple,new tuple> is composed, the 'old tuples' being retrieved from the database. Next, all restrict rules applicable to O are checked. If all of them succeed, O is performed. Then, all the propagate rules applicable to O are invoked, which may lead to recursive calls for triggered data manipulation operations. At the end, in case of success, an SQL/DS 'commit' is performed; otherwise a 'rollback work' takes place.

Note that 'execute' is defined recursively and so propagation may cause arbitrarily long chains of operations to be performed. If any operation in a chain fails, VM/Prolog backtracking causes a rollback along the entire chain. Backtracking also ensures that all branching propagation chains are traversed.

Also note that, because we are using a declarative language, we may add new restrict/propagate rules without having to reprogram 'execute'. In other words, 'execute' cumulatively applies the appropriate restrict/propagate rules to each operation the user submits, thus transforming the operation into a transaction that guarantees the consistency of the data base. This property simplified the design of the tool and will certainly contribute to its evolution.

Another important element of the general component is the *universal restriction rule*:

U. updates on columns corresponding to the key of a table are forbidden.

An application component consists mainly of a number of application-dependent restrict/propagate rules created for each table and taking into consideration the information supplied by the designer in the first phase.

The 'universal' and these application-dependent rules complete our score of measures to preserve integrity. This diversity of integrity-preserving mechanisms is a consequence of the lack in data models and, to an even worse degree, in database management systems of a more comprehensive set of constructs or commands for this purpose.

In what follows we shall look at these specific restrict/propagate rules.

5.3.1. Rules related to is-a and to multi-valued attributes

The simplest restrict/propagate rules have to do with the **is-a** hierarchy and with multi-valued attributes.

Let E and F be entities and suppose that the designer declared that E **is-a** F. Let **E** and **F** be the entity or extended-entitity tables for E and F, respectively. Then,

- the insertion of a tuple into **E** is restricted to the existence of a tuple in **F** with the same key (choice EE1 is fixed);
- the deletion of a tuple from **F** propagates to the deletion of a tuple with the same key from **E**, if such tuple exists (choice EE4 is fixed).

Since the designer may have also declared that G **is-a** F, for another entity G, many tables may be involved both in the restrict and in the propagate rules.

Next, we consider multi-valued attributes. Let E be an entity and A a multi-valued attribute of E. Let **E** and **E_A** be the entity or extended-entity table and the repeating-attribute table for E and A, respectively. Then,

- the insertion of a tuple **a** into **E_A** is restricted to the existence of a tuple **e** in **E** such that the key of **e** is part of the key of **a**;
- the deletion of a tuple **e** from **E** propagates to the deletion of all tuples in **E_A** whose key contains the key of **e**.

The same goes for a relationship table **R** and any associated repeating-attribute table **R_A**.

Both entities and relationships may have several multi-valued attributes and so, again, many tables may be affected by these restrict and propagate rules.

5.3.2. Rules related to general relationships

Next in difficulty, comes the case of relationships implemented by relationship tables (those that are *not* both binary and functional with respect to one participating entity).

Given two tables **T** and **T′** such that the attributes of a key **K** of **T′** are among those of **T**, we say that a tuple **t** in **T** *references* a tuple **t′** of **T′** iff **t**[**K**] = **t′**[**K**].

Let R be a relationship and E one of the entities participating in R. Let **R** and **E** be the relationship table and the entity table for R and E, respectively. Then,

- if R is total with respect to E, the insertion of a tuple **e** in **E** propagates to the insertion into **R** of a tuple **r** referencing **e** (update requirement ERT1); the user will be prompted to supply **r**.
- the deletion of a tuple **e** from **E** is either rejected, if some tuple in **R** references **e** (choice ERI3), or propagates to the deletion of all tuples in **R** that reference **e** (choice ERI4), depending on the designer's decision in the conceptual design phase;
- the insertion of a tuple **r** into **R** is restricted to the existence of a tuple **e** in **E** such that **r** references **e** (choice ERI1 is fixed);
- if R is total with respect to E then the deletion of the last tuple of **R** referencing a tuple **e** of **E** is either rejected (choice ERT2) or propagates to the deletion of **e** from **E** (choice ERT3), depending on the designer's decision in the conceptual design phase.

About this last case a remark is in order. In section 3.4 we pointed out that choice ERI4 forces choice ERT3. The present implementation makes no special provision for this, letting the recursion/backtracking machinery of Prolog take care of chains with two or more operations. However, the next section describes how to enforce a special combination of ERI4/ERT3 involving only one propagation step.

5.3.3. Rules related to binary one-to-n relationships

We finally consider the case of a binary relationship R, with participating entities E and F, such that F is functionally dependent on E in R (that is, to each instance e of E there is at most an instance f of F such that (e,f) is an instance of R). Recall that there will be two tables: an extended-entity table **E** and an entity table **F**. A surprisingly large number of cases must be treated, due to the fact that R is 'embedded' in the extended-entity table **E** (instead of corresponding to a relationship table of its own).

Firstly, consider operations performed on **F**. As for ordinary relationships, we have:

- if R is total with respect to F, the insertion of a tuple **f** in **F** propagates to an update of a tuple **e** in **E** to make it reference **f** (update requirement ERT1); the user will be prompted to supply the key of the tuple **e** to be updated.
- the deletion of a tuple **f** from **F** is either rejected, if some tuple in **E** references **f** (choice ERI3), or propagates to all tuples in **E** that reference **f** (choice ERI4), depending on the designer's decision in the conceptual design phase. In the second option (choice ERI4), two cases must be considered:
  - if R is total with respect to E, the deletion of **f** propagates to the deletion of all tuples in **E** referencing **f** (hence, in this special case, choice ERT3 is fixed).
  - Otherwise, the deletion of **f** propagates to the update of all tuples in **E** referencing **f**, setting to *null* the fields corresponding to the key of F and to the attributes of R.

Consider now operations performed on an extended-entity table **E** representing an entity E and a relationship R. Note that such operations represent either operations on E or operations on R or both, since updates to tuples in **E** that affect fields corresponding to the key of F or to attributes of R are actually updates in R. Moreover, the universal restriction rule (see the beginning of section 5.3) does not forbid the updates on fields corresponding to the key of F since they are not part of the key of **E**.

Then, we have:

(A) Preserving the I-constraint:

- the insertion of a tuple **e** into **E** is rejected if neither all fields corresponding to the key of F and to the attributes of R are all *null*, nor **e** references a tuple of **F** (that is, **e** represents an instance of E and an instance of R and choice ERI1 is fixed);
- the update of a tuple in **E** to a new value **e** is rejected if no value in fields corresponding to the key of F is *null*, but such fields fail to reference a tuple of **F** (choice ERI1);
- an update of a tuple in **E** that sets to *null* one or more fields corresponding to the key of F propagates to an update setting to *null* all other such fields and all the attributes of R;
- the deletion of a tuple **e** from **E** is either rejected, if **e** references a tuple of **F** (choice ERI3), or propagates trivially, since the deletion of **e** eliminates the (single) relationship instance (choice ERI4), depending on the designer's decision in the conceptual design phase;

(B) Preserving the T-constraint when R is total with respect to E

- the insertion of a tuple **e** in **E** is rejected if **e** does not reference a tuple in **F** (update requirement ERT1).
- an update of a tuple **e** in **E** that sets to *null* the fields corresponding to the key of F is:
  - rejected, if the designer's decision is to restrict the deletion of instances of R (choice ERT2), since there is no other tuple **f** in **E** such that **e** and **f** have the same values of the fields corresponding to the key of F;
  - transformed into the deletion of **e** from **E**, if the decision is to propagate deletions (choice ERT3).

(C) Preserving the T-constraint when R is total with respect to F

- the deletion or update of the last tuple of **E** referencing a tuple **f** of **F** is either rejected (choice ERT2) or propagates to the deletion of **f** from **F** (choice ERT3), depending on the designer's decision in the conceptual design phase.

This covers all cases since, by a design decision of the tool, a relationship is total with respect to at most one participating entity.

6. CONCLUSION

The work reported contributes to providing expert help for correct information systems design. Because the process starts with the Entity-Relationship Model, a designer who knows about the application on hand can rely on his intuition, with almost no need to worry about computer-oriented processing. The relational tables induced from this phase are both simple and meaningful. Restrict/propagate rules are generated automatically to govern the behaviour of SQL/DS operations on the database, so as to enforce referential integrity.

The general interface cumulatively applies the appropriate restrict/propagate rules to each operation the user submits, thus transforming the operation into a transaction that guarantees the consistency of the data base. This property simplified the design of the tool and will certainly contribute to its evolution because we may then add new restrict/propagate rules without having to reprogram the interface.

Much remains to be done in two directions at least:

1. refining the strategy;
2. further developing the implementation, especially to improve user interaction.

By applying CHRIS to cases of increasing complexity and by exposing designers with different backgrounds to it, we expect to gain some feed-back necessary to usefully achieve objectives (1) and (2) above.

REFERENCES

[1] V. de Antonellis, G. D. Antoni, G. Mauri, and B. Zonta, Extending the entity-relationship approach to take into account historical aspects of systems, in P. Chen (ed.), Entity-Relationship Approach to Systems Analysis and Design, North-Holland, 1980.

[2] N. Azar, E. Pichat, Translation of an Extended Entity-Relationship model into the universal relation with inclusion formalism, Proc. 5th Int. Conf. on the Entity-Relationship Approach, Dijon, Nov. 1986.

[3] A. Atri, D. Sacca, Equivalence and mapping of database schemes, Proc. 10th Int. Conf. on Very Large Data Bases, Singapore, Sep. 1984.

[4] M. N. Bert, G. Ciardo, B. Demo, et alli, The logical design in the DATAID project: The Easymap System, in A. Albano, V. De Antonellis and A. Di Leva (eds.), Computer-aided database design: the DATAID project, North-Holland, 1985.

[5] P. Bertaina, A. Di Leva, P. Giolito, Logical design in Codasyl and relational environment, in S. Ceri (ed.), Methodology and Tools for Data Base Design, North-Holland, 1983.

[6] M. Bouzeghoub, E. Metais, SECSI: An Expert System Approach for Database Design, Information Processing 86, H.J. Kugler (ed.), North-Holland, 1986 (pp. 251-257).

[7] H. Briand, H. Habrias, J-F Hue, Y. Simon, Expert System for Translating an E-R Diagram into Databases, Proc. 4th Int. Conf. on the Entity-Relationship Approach, Chicago, Oct. 1985.

[8] U. Bussolati, S. Ceri, V. De Antonellis and B. Zonta, Views Conceptual Design, in S. Ceri (ed.) Methodology and Tools for Data Base Design, North-Holland, 1983.

[9] S. Ceri (ed.), Methodology and tools for data base design, North-Holland, 1983.

[10] P. P. Chen - The entity-relationship model: toward a unified view of data - ACM TODS, 1, 1 (1976) 9-36.

[11] M.A. Casanova, R. Fagin, C.H. Papadimitriu, Inclusion Dependencies and their interaction with Functional Dependencies, JCSS Vol.28, No.1 (Fev. 1984).

[12] I. Chung, F. Nakamura, P.P. Chen, A decomposition of relations using the Entitity-Relationship Approach, Entity-Relationship approach to information modelling and analysis, P.P. Chen 1981, North-Holland.

[13] C. J. Date - Relational database: selected writings - Addison-Wesley (1986).

[14] C. J. Date, An introduction to database systems (IV ed.), Addison-Wesley, 1986.

[15] K. P. Eswaran and D. D. Chamberlin, Functional Specification of a Subsystem for Data Base Integrity, Proc. 1st Int. Conf. on Very Large Data Bases (September 1975).

[16] K. P. Eswaran, Specification, Implementation and Interaction of a Trigger Subsystem in an Integrated Data Base System, IBM Research Report RJ1820 (Aug. 1976).

[17] A. L. Furtado - VM/Prolog, etc - adding an expert tool capability to VM/Prolog - technical report CCB041 - IBM Brasil (1986).

[18] P. Hammond and M. Sergot - apes: augmented Prolog for expert systems - Logic Based Systems Ltd. (1984).

[19] M. Lenzerini, G. Santucci, Cardinality constraints in the E-R model, Entity-Relationship Approach to Software Engineering, North-Holland, 1983.

[20] D. Reiner; M. Brodie; G. Brown et alli, The Database Design and Evaluation Workbench (DDEW), IEEE Database Engineering 7:4, Dec. 1984.

[21] C. Rolland, C. Proix, An Expert System Approach to Information System Design, Information Processing 86, H.J. Kugler (ed.), North-Holland, 1986 (pp. 241-250).

[22] SQL/Data System Application programming - doc. IBM SH24-5018-2 (1983).

[23] H. Sakai, A method for entity-relationship behavior modeling, in C. Davis et alli (eds.), Entity-Relationship Approach to Software Engineering, North-Holland, 1983.

[24] H. Sakai, H. Horiuchi, A method for behavior modeling in data oriented approach to system design, in Proc. of the 1st Int. Conf. on Data Engineering, Los Angeles, CA, April 1984 (pp. 492-499).

[25] P. Scheuermann, G. Schiffner, H. Weber, Abstraction capabilities and invariant properties modelling within the Entity-Relationship approach, P.P. Chen (ed.) Entity-Relationship approach to System Analysis and Design, North-Holland, 1980.

[26] VM/Programming in Logic - Program Description and Operations Manual - doc. IBM SB11-6374-0 (1985).