Information Systems Vol. 15, No. 2, pp. 187–195, 1990
Printed in Great Britain. All rights reserved

0306-4379/90 $3.00 + 0.00
Copyright © 1990 Pergamon Press plc

THE CHRIS CONSULTANT—A TOOL FOR DATABASE DESIGN AND RAPID PROTOTYPING

LUIZ TUCHERMAN,[1] MARCO A. CASANOVA,[1] and ANTONIO L. FURTADO[1,2]
[1]Rio Scientific Center, IBM Brasil, P.O. Box 4624, 20.001, Rio de Janeiro, Brazil
[2]Departamento de Informática, Pontificia Universidade Católica do Rio de Janeiro, Rua Marquês de S. Vicente 225, 22.453, Rio de Janeiro, Brazil

(*Received* 10 *October* 1988, *in revised form* 4 *August* 1989; *received for publication* 16 *November* 1989)

Abstract—CHRIS is an expert software tool to help in the design and rapid prototyping 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 and rapid prototyping of information systems containing a database component. As an acronym, CHRIS stands for "Concepts and Hints for Relational Interfaces Synthesis". It incorporates a design method that closely follows what Chris Date describes as "the synthetic approach" [1].

The tool actually consists of two parts: a design tool, which helps the designer specify a database application, and a prototyping tool, which allows the designer or prospective database users to experiment with the design. Figure 1 shows the structure of the tool.

The design tool follows a strategy consisting of three phases:

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.

The logical design phase, where the ER schema is automatically mapped into a normalized relational schema [2]. The schema is complemented by restrict and propagate rules aiming at the enforcement of the integrity constraints defined in the first phase.

The interface generation phase, that synthesizes DDL commands to create the database plus an interface file to be used by the prototyping tool. The interface includes an executable version of the restrict and propagate rules determined at the second phase.

The major contributions of this work are therefore a design tool covering the complete design cycle and a prototyping tool that transforms operations into correct transactions. CHRIS 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.

CHRIS is written in VM/Prolog [3], with the addition of the ETC extension to VM/Prolog [4], whose purpose is to help in the construction of expert systems in general, and which has been largely influenced by APES [5]. The major function provided by ETC is dialogue-management. SQL/DS [6] is used as the target DBMS through the interface between this system and VM/Prolog.

The idea of transforming an extended entity-relationship schema into a relational schema is not new [7–10]. 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 [11].

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 [12–15]. Our approach captures such dynamic behavior in terms of restrict/propagate rules [16, 17], which are then used by an especially designed interface to transform an update operation into a correct transaction, following an idea proposed in [18].

Research on expert tools for database design can be found in [19–22].

The paper is divided as follows. Section 2 describes the specific variation of the entity-relationship model that the tool supports. Sections 3, 4 and 5 present the conceptual design, the logical design and the interface generation phases of the design tool. Section 6 introduces the prototyping tool. Finally Section 7 contains the conclusions and directions for future research.

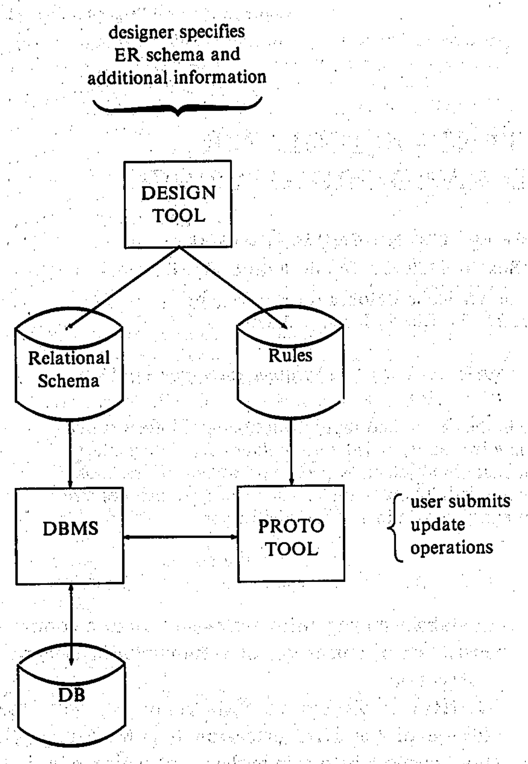

Fig. 1. Architecture of the tool.

2. SUPPORT FOR THE ENTITY-RELATIONSHIP MODEL

2.1. Preliminary remarks

This section describes the specific variation of the ER model [23] that the tool supports.

Briefly, an *ER schema* consists of a set of *entity schemes* and a set of *relationship schemes*. Each scheme has a *name* and a list of *attributes*. Each ER schema also defines a set of *integrity constraints* and indicates how to organize the entity schemes into abstraction hierarchies [18, 24, 25]. The next subsections will discuss these points in detail as well as several restrictions we impose on the description of an ER schema.

The first restriction we impose is that:

- ER1. no two entity or relationship schemes can have the same name.

For simplicity, we will frequently use "entity" (or "relationship") to mean "entity scheme" (or "relationship scheme") in what follows.

2.2. Declaration of attributes

Although the declaration of attributes is part of the declaration of entity or relationship schemes, there are certain common points that we factor out in this section.

Each attribute always has a *type*, which can be viewed as a convention for the representation on a machine or other medium of the attribute value set. For convenience, we restrict ourselves to three of the "physical" types supported by the underlying DBMS, which are INTEGER, FLOAT and CHAR.

The type of an attribute may be qualified in two ways: it may be declared as *single-* or *multi-valued*, and as admitting or not *undefined values*. If an attribute is multi-valued, we assume that it can have zero or more values. We consider zero values as a case of undefined value.

For the entity scheme PERSON, for example, attribute NAME is normally declared to be single-valued (a person has only one name) and not admitting undefined values, whereas TELEPHONE-NUMBER may well be declared to be multi-valued and DRIVER'S-LICENCE as admitting undefined values.

Finally, we require that:

- A1. no two attributes of an entity or relationship scheme may have the same name;
- A2. if the same attribute is associated with two or more schemes, it must have the same type in both uses.

Thus, by (A2), the basic type of an attribute is a global property. However, the properties of being or not multi-valued and of admitting or not an undefined value may differ in each use of the attribute. For example, ADDRESS may be single-valued and always defined for EMPLOYEE, but multi-valued for ENTERPRISE.

2.3. Declaration of entity schemes

The declaration of an entity scheme must specify the name of the scheme and the list of attributes of the scheme, together with their types, as explained in Section 2.2.

In addition, the declaration of an entity scheme E must specify a *key* or *identifier* for E, that 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.

The tool admits only one key per entity.

2.4. Declaration of the **is-a** *relationship*

The tool supports a variation of the **is-a** relationship that is not necessarily a hierarchy.

More precisely, the definition of an ER schema may indicate that an entity scheme E **is-a** a list of entity schemes $F_1, \ldots, F_n$ provided that:

- ISA1. E must be declared after $F_1, \ldots, F_n$;
- ISA2. the identifiers of E, $F_1, \ldots, F_n$ must be the same;
- ISA3. the non-identifying attributes of E, $F_1, \ldots, F_n$ must all have different names.

For each $i = 1, \ldots, n$, all non-identifying attributes of F_i are inherited by E, that is, automatically become attributes of E with the same type, the same single/multiple value property and the same defined/undefined property as in F_i. Moreover, in any consistent state of the ER schema, the set of entities associated with E will always be a subset of the set of entities associated with F_i.

Requirement (ISA1) guarantees that the **is-a** relationship is a partial order (but not necessarily a hierarchy). The other two requirements avoid ambiguities in the inheritance of identifiers and attributes in general and obviate the need to check that the value of an attribute is consistently the same for an entity instance *e* that simultaneously belongs to the sets associated with F_i and F_j.

2.5. Declaration of relationship schemes

The declaration of a relationship scheme must specify a unique name R for the scheme, the list $E_1, \ldots, E_n$ (with $n \geqslant 2$) of *participating* entity schemes, possibly with distinct *roles*, and the list of attributes, together with their types, as explained in Section 2.2. In any consistent state of the ER schema, the set of relationships associated with R will then be a subset of the Cartesian product of the set of entities associated with $E_1, \ldots, E_n$.

We require that all participating entity schemes must have been defined before the relationship scheme.

We let the same entity scheme participate more than once in a relationship scheme, provided that the participations be distinguished by different roles. We consider that a relationship scheme R inherits each attribute A in the identifier of each participating entity scheme E. If E has role P, A is renamed to P_A in R, otherwise it is inherited without renaming. The tool rejects the situation where two or more participants have identifiers with attributes in common; this happens, for instance, when such participants are associated by **is-a**. The use of roles is mandatory in these situations.

A relationship scheme R may also be declared as *total* with respect to at most one participant E to indicate that instances of E cannot exist without participating in at least one instance of R. For example, suppose that, as an enterprise policy, an employee cannot exist unless he is assigned to at least one project. In this case we say that the relationship scheme ASSIGNMENT is total with respect to EMPLOYEE.

Finally, a relationship scheme R may be declared as *functional* with respect to one or more of the participating entity schemes, called the *identifying* participants. This indicates that, if two relationship instances of R involve the same instances of the identifying participants, then the instances of R are actually the same. Moreover, the *identifier* of the relationship scheme is the concatenation of the identifiers (keys) of the identifying participants.

Particular cases of interest are those of binary relationship schemes that are functional with respect to one participant—which are called *one-to-n* relationships (with *n* on the identifying participant side)—or with respect to both—called *one-to-one* relationships. Non-functional binary relationships are said to be *n-to-m*.

To summarize, we require that:

R1. R must have at least two participating entity schemes;

R2. R must be declared after the participating entity schemes;

R2. if an entity scheme participates more than once in a relationship scheme, each participation must be assigned a different role;

R3. all participants must have different names, either roles or not;

R4. if the identifiers of two participating entity schemes have attribute names in common, at least one of the participants must be assigned a role;

R5. R may be declared as total with respect to at most one participating entity scheme.

2.6. Declaration of restrict/propagate options

As discussed in the previous subsections, 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.

The tool considers a special case of this problem, which is how to transform a single insertion, deletion or update operation into a correct transaction with respect to the type of constraints defined in an ER schema. The transformation is based on certain *restrict/propagate options*, specified during the design of the ER schema, and discussed below.

Suppose first that an ER schema declares that E **is-a** F. The options are fixed in this case: the insertion of an instance *e* into E is restricted (i.e. blocked) if *e* is not already an instance of F, and the deletion of an instance *f* from F automatically propagates to the deletion of *f* as an instance of E.

Let us assume that an ER schema declares that R is a relationship scheme over $E_1, \ldots, E_n$. An obvious integrity constraint, call it the *incidence constraint*, is that an instance of R cannot exist unless the participating instances from each E_i also exist. To preserve this constraint, the ER schema must indicate whether the insertion of an instance $(e_1, \ldots, e_n)$ of R must be restricted (i.e. blocked), if e_i is not an instance of E_i, for each i in $[1, n]$, or propagate to the insertion of e_i into E_i, if e_i is not an instance of E_i. Likewise, the ER schema must also indicate whether the deletion of an instance e_i of E_i must be blocked, if an instance $(f_1, \ldots, f_n)$ of R with $f_i = e_i$ exists, or be followed by the deletion of all instances $(f_1, \ldots, f_n)$ of R with $f_i = e_i$.

Suppose now that an ER schema declares that R is *total* over E_i. In this case, a single possibility for transforming the insertion of a new instance e_i of E_i presents itself: it must be followed by the insertion of an instance $(f_1, \ldots, f_n)$ of R such that $f_i = e_i$. This means that, if a correct transaction contains an insertion of a new instance e_i into E_i, it must also contain an insertion into R. But which instances $f_1, f_2, \ldots, f_{i-1}, f_{i+1}, \ldots, f_n$ should we take? Unless some criterion has been chosen, the only option we have is to consult the person performing the insertion of e_i. The situation would in fact be more complex if a relationship sheme were allowed to be total with respect to more than one entity scheme, which for simplicity the tool explicitly forbids. To disambiguate the treatment of the deletion of an instance $(e_1, \ldots, e_n)$ of R, the ER schema must indicate whether it must be restricted, if there is no other instance $(f_1, \ldots, f_n)$ of R with $f_i = e_i$, or propagate to the deletion of e_i from E_i, if there is no other instance $(f_1, \ldots, f_n)$ of R with $f_i = e_i$.

The decisions related to the incidence constraint are not independent from those related to totality. For example, if the designer has chosen that deletion of an instances e_i of E_i participating in a relationship R that is total with respect to E_i should be blocked, the decision to also block the deletion of the last instance r of R where e_i participates leads to a situation that is not usually intended: once created, e_i can never be deleted.

Finally, in all other situations, such as violations of attribute types or uniqueness of identifiers, the operation is blocked.

3. CONCEPTUAL DESIGN PHASE

3.1. Outline of the 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.

As a result of the dialogue a *log* is created, consisting of a number of Prolog clauses. When the designer initiates the definition of each entity or relationship, control is transferred to a program—called *scheduler*—which engages him in a dialogue to supply the various components of the definition, according to a *definition order*. Moreover, the dialogue is driven by certain *rules* that validate the designer's answers (*valid_answer*), declare that only one answer should be supplied in specific cases (*unique_answer*), prompt the designer to supply new answers (*complete_answer*) or automatically fire (*trigger*) new internal operations as a result of an answer.

The following figure summarizes the structure of this phase, where the single arrows denote program invocation and the double arrow denotes output [26]:

$$\begin{tikzcd} & \text{scheduler} \arrow[d] & \\ \text{rules} & \text{interactive facility} \arrow[l] \arrow[r, Rightarrow] & \text{log} \end{tikzcd}$$

In addition to commands to design entity or relationship schemes, described in detail in the next two subsections, at any point during a session, the designer may also issue the command **keep(F)** to save the current status of the design in a file **F**. Conversely, the designer may reinitiate a session saved in **F** by issuing the command **restore(F)**. The **drop(X)** command deletes all clauses related to an entity or relationship **X**, as well as all clauses related to other entities or relationships defined after **X**, since their definitions may have become incorrect due to the removal of **X** (of course, more selective redesign features are desirable). Finally, before replying to a query posed to him, the designer may want to inspect previous definitions recorded in the log, which is provided by the **log** command through a menu selection.

3.2. Defining an entity

The tool offers a single command, **entity(E)**, to define an entity scheme, as well as the **is-a** relation.

We now describe the successive definition steps determined by the entity scheduler and the rules that drive the dialogue. The kind of reply expected from the designer is indicated in each case. In this dialogue, "**Q**", stands for a question posed by the tool and "**A**", the expected answer from the designer.

Q1: If other entities have been previously defined, is E a sub-class of one or more of them?
A1: from menu.
Q2: Which are the attributes of E?
A2: name of attribute.

For each such attribute A that has not been inherited from any super-class:

Q3: What is the type of A?
A3: from menu. If the type selected is character string, enter an integer determining the maximum string length.
Q4: Does A admit repeating values?

A4: yes or no. If yes, it is assumed that A can have 0 or more values, i.e. the value of A can be undefined.

Q5: If A does not admit repeating values, can the value of A be undefined?

A5: yes or no.

Q6: Which are the identifying attributes of E?

A6: from menu.

Notes—we remind that:

(1) Names of attributes are unique, and so attribute A of an entity E is the same as A of another entity F or relationship R. Accordingly, the type of A is asked only when A is first introduced. Yet the properties of being multi-valued and/or admit an undefined value can differ for A in E and F (or R).

(2) Entities admit a single identifier, which may be simple or composite. Therefore when more than one identifying attribute is selected, a composite attribute is assumed.

The rules that drive the dialogue for the definition of entities are:

(1) valid_answer rules

Question	Test applied
Q1:	only entities previously defined are displayed in the menu; if the designer selects more than one super-class, they must all have the same identifiers and must all have distinct non-identifying attributes.
Q3:	only INTEGER, FLOAT or CHAR(⟨number of characters⟩) are displayed in the menu.
Q6:	only attributes of E that are not multi-valued and cannot be undefined are displayed in the menu.

(2) trigger rules

Question	Actions taken
Q1:	the attributes of super-class F are inherited by E; they will be multi-valued and/or accept null values if they are so with respect to F. The identifiers of E are exactly those of F.
Q4:	multi-valued attributes are automatically specified as undefined.

(3) unique_answer rules

Question	Test applied
Q3:	only one type can be specified per attribute.

(4) complete_answer rules

Question	Test applied
Q6:	every entity must have at least one identifying attribute.

3.3. Defining a relationship

The tool has a single command, **relationship(R)**, to define a relationship scheme.

Again, we now describe the successive definition steps determined by the relationship scheduler and the rules that drive the dialogue:

Q1: Which are the participants P of R?

A1: name of an entity or name of a role.

For each participant P

Q2: If P is the name of a role, then enter the name of the entity for which P is the role in R.

A2: from menu.

Q3: If R has not already been declared total with respect to some other participant Q, is R total with respect to P?

A3: yes or no. If the answer is yes:

Q4: Does the deletion of the last instance of R that references an instance of P propagate to the deletion of this instance of P?

A4: yes or no.

Q5: Does the deletion of an instance of P propagate to the deletion of the instances of R that reference that instance of P?

A5: yes or no.

Q6: Which participants are identifiers of R?

A6: from menu.

Q7: Which are the attributes of R, besides the identifiers of the participants?

A7: name of attribute.

For each such attribute, A

Q8: What is the type of A?

A8: from menu. If the type selected is character string, enter an integer determining the maximum string length.

Q9: Does A admit repeating values?

A9: yes or no.

Q10: If A does not admit repeating values, can the value of A be undefined?

A10: yes or no.

[see also note (1) following the entity definition dialogue].

The rules that drive the dialogue for the definition of relationships are:

(1) valid_answers rules

Question	Test applied
Q1:	only previously defined entities can be specified, and no two participants of R can have identifiers with attributes in common (the use of roles can solve this problem as explained in Section 2.5).
Q2:	only previously defined entities are displayed in the menu.
Q6:	only participants are displayed in the menu.
Q8:	only INTEGER, FLOAT or CHAR(⟨number of characters⟩) are displayed in the menu.

(2) unique_answer rules

Question	**Test applied**
Q8:	only one type can be specified per attribute.

(3) trigger rules

Question	**Actions taken**
Q1:	define the attributes of the identifiers of participating entities with no assigned roles as attributes of the relationship.
Q2:	for each attribute A of the identifier of an entity participating through a role P in R, define P_A as an attribute of R.
Q6:	the concatenation of the identifiers of identifying participating entities becomes the identifier of the relationship.
Q9:	multi-valued attributes are automatically specified as undefined.

(4) complete_answer rules

Question	**Test applied**
Q1:	relationship R must have at least two participants.
Q6:	relationship R must have at least one identifying participant.

4. LOGICAL DESIGN PHASE

To map entities and relationships into relational tables, we start with the simple strategy of assigning to each entity E a table, also named E, and to each relationship R, a table R. The columns of tables E or R correspond to the respective attributes and the key of the table corresponds to the identifier of the entity or relationship. The presence of **is-a** introduces a simplification: if E **is-a** F, then table E must keep only the attributes of the identifier (which is the same as that of F) and the attributes not inherited from F.

However, the first normal form requirement conflicts with the notion of multi-valued attributes. Such attributes are placed in separate *repeating-attribute* tables, together with the identifiers of the associated entity or relationship.

Binary one-to-n relationships also receive a special treatment. Consider one such relationship R between E and F, where E is the identifying participant. Then the identifier of R is exactly that of E, and hence the keys of tables E and R are the same. Then, we can compress tables E and R into a single table, reducing the number of original tables, without violating third normal form. This led to the *extended-entity* table concept, whereby a single table represents both an entity and one (or more) binary one-to-n relationships, together with their attributes.

The adoption of extended-entity tables turned out to introduce far more complexity than we anticipated. We decided not to use them when: (1) the identifying entity E particpates through a role; (2) there is more than one binary one-to-n relationship, say R1 and R2, between the same participants E and F, except if F is distinguished by roles in R1 or R2.

Depending on the type of a table and on the options taken by the designer during the dialogue at the conceptual design phase, the tool also attaches different restrict and propagate rules to the table. For example, if an entity scheme E participates in a general relationship R, the deletion of a tuple *e* from E may propagate to or be restricted by the existence of a tuple *r* in R that represents the participation of *e* in the relationship. If R is total with respect to E, the insertion of a tuple *e* in E necessarily propagates to the insertion into R of a tuple referencing *e*, to be supplied by the user.

Restrict and propagate rules break down into a diversity of cases for extended-entity tables. One should recall that such tables are not limited to representing the instances of an entity since they embed one (or more) relationships. Thus operations on their tuples may be directed either to the entity or to a relationship, and may affect both in some situations. Another source of complexity is that the (foreign) keys denoting the other entities may be null if the relationship is not total, which requires that rules be attached not only to insertions and deletions but also to updates, so that these values are handled correctly.

To give an idea of how these rules are formed from the information gathered at the conceptual design phase, consider the definition of a one-to-n relationship WORKS between EMPLOYEE and DEPARTMENT and suppose that the designer replies (see Section 3.3):

(1) YES to questions **Q3** and **Q4**, when EMPLOYEE is indicated as a participant of WORKS;

(2) YES to **Q5**, when DEPARTMENT is indicated as a participant of WORKS;

Together, the replies in (1) mean that WORKS is total for EMPLOYEE and that the deletion of an employee-department pair (e, d) from WORKS propagates to the deletion of the employee *e* from EMPLOYEE, if *e* does not work for any other department. Furthermore, the reply in (2) means that the deletion of a department *d* propagates to the deletion of all entries in WORKS that reference *d*.

Suppose now that, at the logical design phase, EMPLOYEE and WORKS are collapsed into a single extended-entity table, also called EMPLOYEE, and that DEPARTMENT is mapped into an ordinary entity table with the same name. Then, a propagation rule is created and stored as the following Prolog clause:

```
propagate(rd1,delete,department,employee)
```

where the parameters delete and department indicate that the rule is applicable when a tuple *t* of

DEPARTMENT is deleted; elsewhere in the system it is established that **rd1** means that the propagation action is the deletion of all tuples of the other table, **EMPLOYEE** in this case, referencing *t*.

The previous clause is simply displayed to the designer in the format:

delete department → delete employee

Now, if the reply to **Q3** were NO, a different kind of propagate rule would be created, whose display format is:

delete department → nullify reference in employee

Finally, if the reply to **Q5** were NO, regardless of the replies to **Q3** and **Q4**, the result would be a restrict rule forbidding the deletion of departments with employees, displayed in the format:

delete department / no reference in employee

To conclude, the **structure** command displays the relational tables that represent the entity-relationship definitions and the **behavior** command displays the restrict and propagate rules. Finally, the command **schema(F)** exhibits both the tables' organization and the rules and stores them in file **F**.

5. THE INTERFACE GENERATION PHASE

The mapping from relational tables to SQL/DS tables is of course immediate, since SQL/DS follows closely the relational model. However, a number of remarks must be made.

The order of columns is immaterial in the relational model, but in an implementation the order is significant in many cases. The SQL/DS columns of a table **T** will follow the definition order coming from the entity-relationship phase.

To enforce keys in SQL/DS, the tool defines indexes on keys with the "unique" option.

Whenever undefined values are excluded for an attribute **A**, the respective column is declared with the "not null" option in the "create table" command. An exception occurs if **A** denotes an attribute of a relationship **R** that is not total with respect to **E** and both **E** and **R** are mapped into an extended-entity table. In this case, the value of **A** in a tuple of the table will be "null" if and only if the instance of **E** represented by the tuple does not currently participate in **R**. In fact, in such tuple all attributes originating from **R** must be "null".

As the tool creates a table, it also creates restrict and propagate rules associated with executable VM/Prolog code and stores them in an interface file indicated by the designer, which is used for monitored execution by the prototyping tool. The executable code created is stored as an additional parameter in the clauses recording the rules, and takes into consideration which tables are involved and what is to be done. For restrict rules, this may include SQL/DS select commands. For propagate rules, insert, delete and update commands may also be included, to be invoked from the recursive **execute1** metapredicate (see Section 6). In the example of Section 5, the code of the propagate rule displayed as

delete department → nullify reference in employee

should perform an update on the appropriate **EMPLOYEE** tuples to set to null the values of the fields corresponding to the key of the **DEPARTMENT** tuple deleted and to the attributes (if any) of the **WORKS** relationship.

For the execution of updating commands the tool provides different error-checking methods. In some cases it uses the SQL return codes, but generally most of the checks are done via the restrict/propagate rules.

These rules avoid that key values be updated, prevent the insertion of tuples with less values than specified in the table definition, control nulls in the case mentioned before for extended-entity tables, where the "not null" option would not apply, and—as their most important purpose—guarantee the referential integrity requirements arising from **is-a**, connections with repeating-attribute tables, total and partial relationships, etc.

The physical creation of the SQL/DS tables, plus indices with the "unique" option, and of the file containing Prolog predicates corresponding to the restrict and propagate rules is accomplished by the command **implement(F,D)**, where **F** is the name of a file wherein the predicates for the restrict and propagate rules are stored, and **D** is the database space where the tables and indexes will be kept. Besides the restrict and propagate rules, other useful information is stored in file **F**, especially the record of the "create table" and "create index" operations executed.

6. THE PROTOTYPING TOOL

If **F** is the application interface file created at the interface generation phase, the user initiates a prototyping session by entering **interf F**.

This initiates a VM/Prolog session, where the database can be updated and queried, monitored by the restrict and propagate rules in **F**. Besides **F**, the command **interf F** loads a file containing generic commands for the monitored manipulation of databases and the two *universal restrict rules*:

ri0. Forbids insertion in any table of tuples whose number of values is not the same as the number of columns of the table.
ru0. Forbids update of keys of any table.

The major command of the prototyping tool is **execute(O)**, where **O** is either an SQL "select" command or an SQL "insert", "delete" or "update" command. Its main purpose is to handle the execution of "insert", "delete" and "update" operations, monitored by the restrict and propagate rules. Before executing these operations, the user may want to see the "create table" command originating the table in order to correctly write the sequence of values of the "insert" operation. The associated "create index" command may also be inspected. The execution of SQL commands may be traced, which is especially useful to follow the chains of commands induced by the propagate rules.

For update operations, **execute** calls **execute1** and, in case of success, forces an SQL "commit", otherwise it issues a "rollback". In turn, **execute1** proceeds along the followign steps: (1) identification of the operation; (2) analysis of the tuples affected; (3) test of each applicable restriction; (4) execution of the operation itself; (5) call to **execute1** for each applicable propagation. The last step may cause the execution of a chain of operations, induced by the recursive structure of **execute1**. So, the "commit" or "rollback" command issued by **execute** refers to the entire chain of operations. If the trace option of the tool is "on" (which is the default), each operation is displayed as it is executed.

7. CONCLUSIONS

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 behavior of update operations on the database, so as to enforce certain classes of integrity constraints.

The prototyping tool cumulatively applies all appropriate restrict/propagate rules to each operation the user submits, thus transforming the operation into a transaction that guarantees the consistency of the database. This property simplified the design of the tool and will certainly contribute to its evolution because it allows adding new restrict/propagate rules without reprogramming.

The scope of the design tool was deliberately limited in order to have a first version available in a reasonably short time. Yet, the implementation turned out to be quite complex. Some of the limits affect the design strategy itself, the most important being:

(1) Attribute types are simply those of the underlying DBMS.

(2) 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.

(3) A relationship can be declared total with respect to no more than one participating entity.

(4) At the conceptual design phase, the designer can choose between restriction or propagation only in certain cases.

The implementation was likewise simplified. Only at the conceptual design phase the prototype establishes a dialogue with the designer. 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 propagated insertions and updates.

Much remains to be done in at least two directions: refining the strategy and further developing the implementation, especially to include more advanced features to correct or modify the design of applications [26] and to improve user interaction. By applying CHRIS to cases of increasing complexity and by exposing it to designers with different backgrounds, we expect to gain the necessary feed-back to usefully achieve the two objectives above.

REFERENCES

[1] C. J. Date. *Relational Database: Selected Writings*. Addison-Wesley, New York (1986).
[2] C. J. Date. *An Introduction to Database Systems*, 4th edn. Addison-Wesley, New York (1986).
[3] VM/Programming in Logic—program description and operations manual. Doc. IBM SB11-6374-0 (1985).
[4] A. L. Furtado. VM/Prolog, etc.—adding an expert tool capability to VM/Prolog. Technical Report CCB041, IBM Brasil (1986).
[5] P. Hammond and M. Sergot. APES: *Augmented Prolog for Expert Systems*. Logic Based Systems, Ltd (1984).
[6] SQL/Data System application programming. Doc. IBM SH24-5018-2 (1983).
[7] N. Azar and E. Pichat. Translation of an extended entity-relationship model into the universal relation with inclusion formalism. *Proc. 5th Int. Conf. Entity-Relationship Approach*, Dijon (1986).
[8] A. Atri and D. Sacca. Equivalence and mapping of database schemes. *Proc. 10th Int. Conf. Very Large Data Bases*, Singapore (1984).
[9] M. N. Bert, G. Ciardo, B. Demo *et al.* The logical design in the DATAID project: the Easymap system. In *Computer-aided Database Design: The DATAID Project*. (Edited by A. Albano, V. De Antonellis and A. Di Leva). North-Holland, Amsterdam (1985).
[10] I. Chung, F. Nakamura and P P. Chen. A decomposition of relations using the entity-relationship approach. In *Entity-Relationship Approach to Information Modelling and Analysis* (Edited by P. P. Chen) North-Holland, Amsterdam (1981).
[11] P. Bertaina, A. Di Leva and P. Giolito. Logical design in Codasyl and relational environment. In *Methodology and Tools for Data Base Design* (Edited by S. Ceri) North-Holland, Amsterdam (1983).
[12] 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 *Entity-Relationship Approach to Systems Analysis and Design* (Edited by P. Chen). North-Holland, Amsterdam (1980).
[13] S. Ceri (Editor) *Methodology and Tools for Data Base Design*. North-Holland, Amsterdam (1983).

[14] H. Sakai. A method for entity-relationship behavior modeling. In *Entity-Relationship Approach to Software Engineering* (Edited by C. Davis *et al.*). North-Holland, Amsterdam (1983).

[15] H. Sakai and H. Horiuchi. A method for behavior modeling in data oriented approach to system design. *Proc. 1st Int. Conf. Data Engineering*, Los Angeles, pp. 492–499 (1984).

[16] K. P. Eswaran and D. D. Chamberlin. Functional specification of a subsystem for data base integrity. *Proc. 1st Int. Conf. Very Large Data Bases* (1975).

[17] K. P. Eswaran. Specification, implementation and interaction of a trigger subsystem in an integrated data base system. IBM Research Report RJ1820 (1976).

[18] P. Scheuermann, G. Schiffner and H. Weber. Abstraction capabilities and invariant properties modelling within the entity-relationship approach. In *Entity-Relationship-Approach to System Analysis and Design*, (Edited by P. P. Chen). North-Holland, Amsterdam (1980).

[19] M. Bouzeghoub and E. Metais. SECSI: an expert system approach for database design. In *Information Processing 86* (Edited by H. J. Kugler), pp. 251–257. North-Holland, Amsterdam (1986).

[20] H. Briand, H. Habrias, J-F Hue and Y. Simon. Expert system for translating an E–R diagram into databases. *Proc. 4th Int. Conf. Entity-Relationship Approach*, Chicago (1985).

[21] D. Reiner, M. Brodie, G. Brown *et al.* The database design and evaluation workbench (DDEW). *IEEE Database Engng* 7(4) (1984).

[22] C. Rolland and C. Proix. An expert system approach to information system design. In *Information Processing 86* (Edited by H. J. Kugler), pp. 241–250. North-Holland, Amsterdam (1986).

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

[24] U. Bussolati, S. Ceri, V. De Antonellis and B. Zonta. Views conceptual design. In *Methodology and Tools for Data Base Design* (Edited by S. Ceri). North-Holland. Amsterdam (1983).

[25] M. Lenzerini and G. Santucci. Cardinality constraints in the E-R model. In *Entity-Relationship Approach to Software Engineering*. North-Holland, Amsterdam (1983).

[26] A. L. Furtado, M. A. Casanova and L. Tucherman. A framework for design/redesign experts. *Proc. 1st Int. Conf. Expert Database System*, pp. 313–328 (1986).