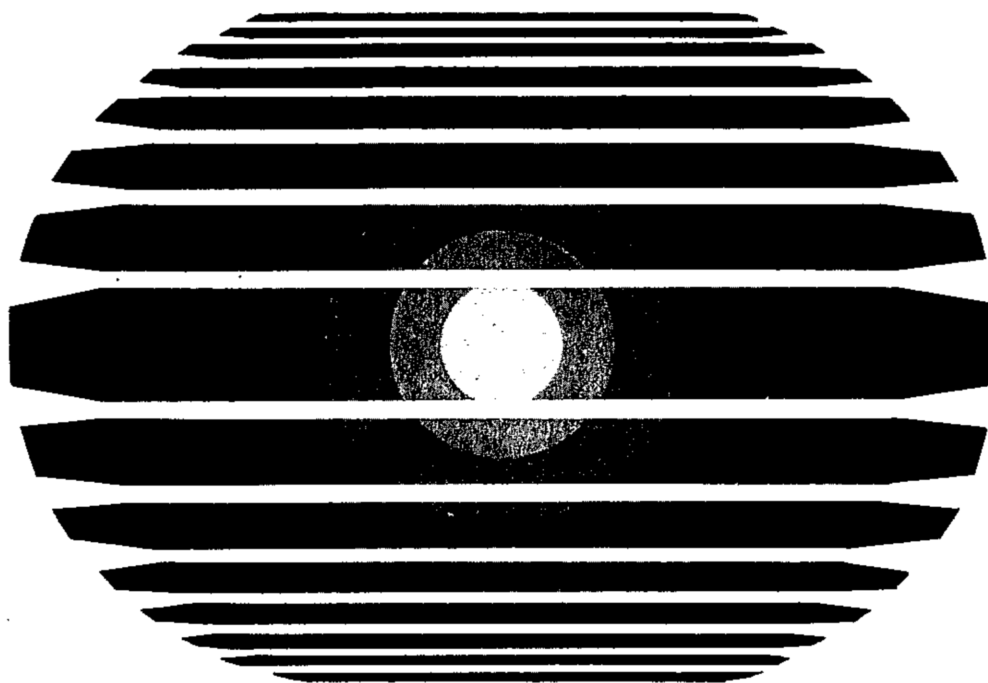


Rio Scientific Center

Rio de Janeiro

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

CCR061 October-88

IBM Brasil

044

TECHNICAL REPORT

CCR061 DATE: OCTOBER-88

THE CHRIS CONSULTANT - A TOOL FOR DATA BASE DESIGN AND RAPID PROTOTYPING

by

Antonio L. Furtado*
Marco A. Casanova
Luiz Tucherman

Rio Scientific Center
IBM Brasil
P.O Box 4624
20.071, Rio de Janeiro, RJ

* On leave from the Pontíficia Universidade Católica do Rio de Janeiro

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 system 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 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.

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;

the interface generation phase, that synthesizes DDL commands to create the database 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 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 [26], with the addition of the ETC extension to VM/Prolog [17], whose purpose is to help in the construction of expert systems in general, and which has been largely influenced by APES [18]. The major function provided by ETC is dialogue-management. SQL/DS [22] 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 [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 an especially 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 describes the specific variation of the Entity-Relationship Model that the tool supports. Section 3 outlines the design strategy the tool implements and the overall structure of the tool. Sections 4, 5 and 6 present the conceptual design, the logical design and the interface generation phases of the design tool. Section 7 introduces the prototyping tool. Finally, section 8 contains the conclusions and directions for future research. Appendix I and II show a complete example of the use of the tool and Appendix III lists the software and hardware requirements of the tool.

2. SUPPORT FOR THE ENTITY-RELATIONSHIP MODEL

2.1 Preliminary Remarks

This section describes the specific variation of the ER Model [10] 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 [8,19,25]. 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-LICENSE 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 $F_1,...,F_n$, provided that:

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

For each $i=1,...,n$, 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 $E_1,...,E_n$ (with $n \geq 2$) 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 $E_1,...,E_n$.

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 assume that an ER schema declares that R is a relationship scheme over $E_1,...,E_n$. 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 $(e_1,...,e_n)$ 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 $(f_1,...,f_n)$ of R with $f_i = e_i$ exists, or be followed by the deletion of all instances $(f_1,...,f_n)$ 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 $(f_1,...,f_n)$ 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 $f_1,f_2,...,f_{i-1},f_{i+1},...,f_n$ 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 scheme 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 $(e_1,...,e_n)$ of R, the ER schema must indicate whether it must be restricted, if there is no other instance $(f_1,...,f_n)$ of R with $f_i = e_i$, or propagate to the deletion of e_i from E_i, if there is no other instance $(f_1,...,f_n)$ 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. GENERAL DESCRIPTION OF THE TOOL

3.1. Structure of the tool

The tool consists of two separate programs:

1. The design tool, which helps the designer specify a database application along three phases, to be described below;

2. the prototyping tool, which allows the designer and/or prospective database users to experiment with the design.

The design tool follows a 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 extended entity-relationship schema is automatically mapped into a relational schema. The schema is complemented by restrict and propagate rules aiming at the enforcement of the integrity constraints defined in the first phase.

3. The interface generation phase, that synthesizes and executes 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 programs for the design and the prototyping tools are written in VM/Prolog, extended with the ETC query-the-user capability and the PROWIN front-end, which provides the windows environment. In both programs SQL/DS is used, across the PROSQL interface.

The following figure summarizes the architecture of the tool:

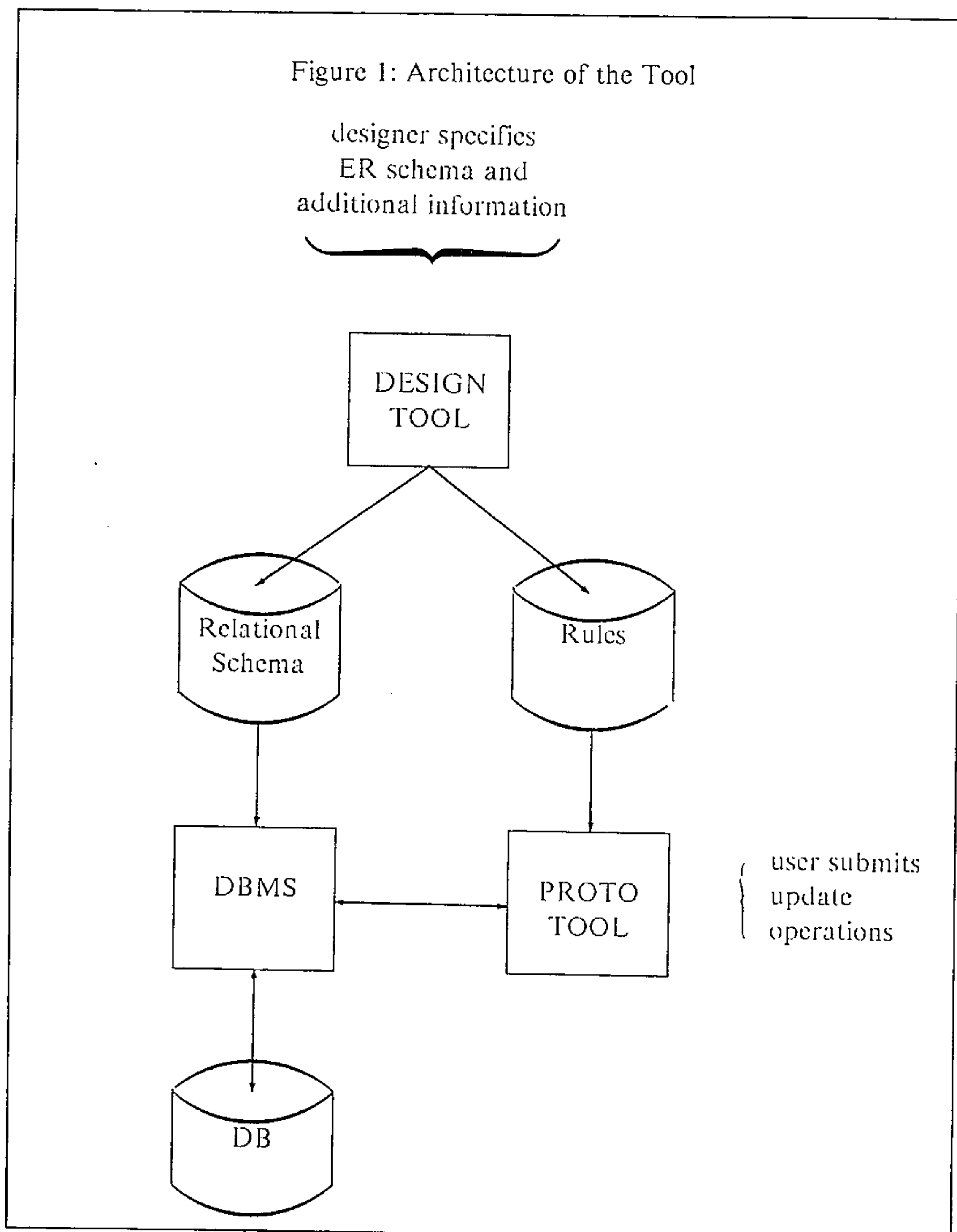


Figure 1: Architecture of the Tool

The scope of the tool was deliberately limited in order to have a first version 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 by the tool 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.

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

- 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 propagate insertions and updates.

3.2. Environments to run CHRIS

The design tool works in two environments:

- the "normal" VM/Prolog environment

- the windows environment, provided by the PROWIN front-end

The windows used are listed below with their function:

EXEC where the designer enters his commands (sections 4, 5 and 6).

PROMPT where the system poses questions to the designer.

INPUT where the designer enters answers to questions posed by the system or, before entering an answer, asks for information useful to formulate the answer.

FORMS a pop-up window, with two functions:

- to provide a menu from which the designer must select his answer to certain questions posed by the system;

- to display the log of the dialogue, which is one of the ways whereby a designer can search for information before answering a question.

This window overlaps the INPUT window.

GIVEN where the system places previous designer' answers when it asks for further answers.

MESSAGES where the system places messages related to "valid_answer" and "complete_answer" rules and replies to questions asked by the designer, for example to list information contained in the log.

COMM a pop-up window, where trapped network system messages are displayed (a function handled by the MSGGET package, used by PROWIN); this window overlaps the MESSAGES window.

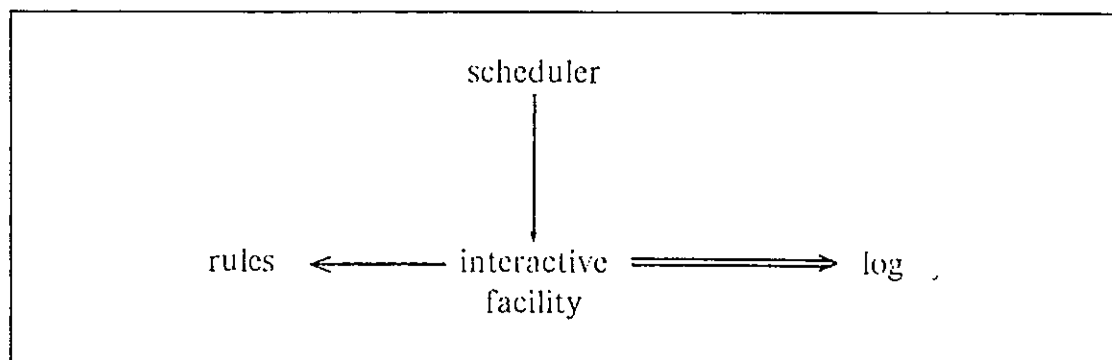
When describing the design tool, in sections 4, 5 and 6, we will assume the windows environment. A short description concerning the normal environment is provided at the beginning of the Appendix I.

The prototyping tool works only on the normal VM/Prolog environment.

4. CONCEPTUAL DESIGN PHASE

4.1 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. 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.



As a result of the dialogue a *log* is created, consisting of a number of Prolog clauses. When, from the 'EXEC' window, 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 the following types of *rules*:

valid_answer checks if an answer supplied by the designer is acceptable; if it is not acceptable, the message '***rejected***' is issued; the message '***repetition***' is issued when the same value

is entered twice; recall that in some cases, the valid answers are those in a menu.

unique_answer determines that a question admits only one answer, so that the system does not continue to prompt the designer for more replies.

trigger adds to the log further clauses derived as a result of an answer.

complete_answer establishes whether the designer's answers are sufficient, when he signals the end of his replies; if not, a message is issued and the designer continues to be prompted for answers. The message is part of the complete_answer rule.

Moreover, there are template-rules to translate the predicate syntax into a syntax closer to natural language, to be used in the dialogue. These rules are:

is_template for queries to the designer that admit yes/no answers

which_template for all other queries to the designer

read_as to list clauses belonging to the log

4.2. Commands

To initiate the design session, first enter from the CMS environment:

chris

to load the CHRIS tool, as well as the various systems used by CHRIS. You are placed in the VM/Prolog environment. Next enter

design.

to enable the windows environment, in which the commands below can be executed from the 'EXEC' window (do not forget the stop mark at the end):

entity(E). To start the definition of an entity E.

relationship(R). To start the definition of a relationship R.

keep(F). F is the name of a file where the log is to be saved before terminating a session. It is a good practice to perform "keep(F)" often during the session, as a "check-point" policy; each time F is overwritten with the contents of the new state.

restore(F). F is the name of a file created in a previous session. If the current session is a continuation of the session recorded in F, this command should be executed - only once - immediately after entering the windows environment via "design".

drop(X). To delete all clauses related to an entity or relationship X as well as all other entities and relationships defined after X (see the "defs" command below).

log. The names of the predicates in the log are displayed in the 'FORMS' window. The designer can select one or more of them to be listed. The cursor is moved up by PF1 and down by PF2; when the cursor is over the selected item, hit the 'enter' key. To terminate, select *end*.

like(E,F). To declare (and add to the workspace) the fact that some element E is_like another element F (and vice-versa).

defs. To enumerate the entities and relationships thus far introduced, in the order they were defined.

fin. To terminate a session.

During a dialogue, to define an entity or relationship, certain commands can be executed from the 'INPUT' window before answering a question posed by CHRIS. Some commands correspond to keywords and others to predicates. If the question is being asked through a menu, first select *user* and then enter the command; otherwise enter the command directly.

The commands include "like" and "log", already mentioned, and:

end. Indicates that there are no more answers to a query (possibly no answers at all). Inside menus, this keyword is written *end*.

valid. The valid_answer rule attached to the query is displayed.

alike. Instances of a clause p for elements declared to be like those in p are displayed.

user When a menu of permissible answers is displayed, this keyword must be selected if the designer wants to execute some command or enter some keyword before selecting from the menu.

drop. To abort the definition of an entity or relationship; all clauses related to the entity or relationship are deleted. A "user error" is forced to interrupt the scheduler; after the designer hits the "enter" key, the cursor returns to the 'EXEC' window.

4.3. 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 CHRIS and "**A**", the expected answer from the designer.

4.3.1 Scheduler

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 multivalued and/or admit an undefined value can differ for A in E and in 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.

4.3.2 Rules

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

- 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.

- 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.

- unique_answer rules

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

- complete_answer rules

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

4.4. 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:

4.4.1 Scheduler

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)

4.4.2 Rules

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

- valid_answer 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 same identifiers will occur for instance in entities associated by is-a; 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.

- unique_answer rules

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

- 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 P and 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.

- 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.

4.5. Predicate Clauses added to the Log

- For an entity E:

is_entity(E)	E is defined as an entity
is_a(E,F)	entity E is a sub-class of entity F
attribute(E,A)	A is an attribute of E
type(A,T)	the type of A is T
repeating(E,A)	attribute A is multi-valued for entity E
can_be_undef(E,A)	attribute A can be undefined for entity E
identifier(E,K)	attribute K is part of the identifier of E

- For a relationship R:

is_relat(R)	R is defined as a relationship

participant(R,E)	entity or role E participates in R
role(R,E,F)	E is a role of entity F in R
is_total(R,E)	relationship R is total with respect to entity E
del_rel_prop(R,E)	the deletion of instances of relationship R propagate to the deletion of the instances of entity E that no longer participate in R
del_ent_prop(R,E)	the deletion of instances of entity E propagates to the deletion of the participations of E in R
id_participant(R,E)	entity E is an identifying participant of R
attribute(R,A)	A is an attribute of relationship R
type(A,T)	the type of A is T
repeating(R,A)	attribute A is multi-valued for relationship R
can_be_undef(R,A)	attribute A can be undefined for relationship R

5. LOGICAL DESIGN PHASE

5.1 Outline of the 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 multivalued 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:

- the identifying entity E participates through a role;
- 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.

5.2. Commands

At the end of the conceptual design phase (or even after a dialogue to define an entity or relationship), the following commands can be used, from the 'EXEC' window, to show the relational tables into which the entity-relationship definitions are mapped, as well as the restrict and propagate rules that should discipline the updates to these tables. The commands are:

structure. to display the relational tables.

behaviour. to display the restrict and propagate rules.

schema(F). to display both the tables' organization and the rules and then to stores them in file F.

fin. to terminate a session

5.3. Tables

There are three predicates to define tables:

table(N,T)	where N is the name of a table and T is its type (see below);
attr_table(N,A)	where N is the name of a table and A is one of its columns;
key(N,A)	where N is the name of a table and A is part of its key.

The three predicates above map an entity-relationship schema into a relational schema. The mapping can best be described according to the types of the relational tables:

- entity tables - for an entity E

table name	E
columns	non-repeating attributes of E, except the non-identifying attributes inherited via is_a
key	sequence of identifiers of E

- relationship tables - for a relationship R

table name	R
columns	identifiers of the participants of R and non-repeating attributes of R
key	sequence of the identifiers of identifying participants of R

- repeating-attribute tables - for each multi-valued attribute A of C, where C can be an entity or a relationship

table name	C_A
columns	identifiers of C and the attribute A
key	sequence of the identifiers of C and A

- extended-entity tables - for binary relationships R between entity E and participant P (entity or role of entity, possibly a role of E); where E is the identifying participant

table name	E
columns	non-repeating attributes of E, identifiers of P and non-repeating attributes of R

key sequence of identifiers of E

Note: Any number of binary relationships R1, R2, ... ,Rn between E and participants P1, P2, ... ,Pn, such that E is the identifying participant in all Ri, are collapsed in the same table E. However, no two relationships having an attribute in common, besides the identifiers of E, are collapsed. Assigning different roles to participants may avoid the problem.

5.4. Restrict and Propagate Rules

The restrict and propagate rules are displayed, respectively, with the formats:

<operation> <table> / <restriction condition>

<operation> <table> -> <propagation action>

where <operation> is insert, delete or update. The translation from predicate syntax into the notation above is determined by "res_read" and "prop_read" template rules. The parameters of the "restrict" and "propagate" predicates are shown below. Details on the type parameters RN and PN will be given later in this section.

- restrict(RN,O,T,V) - where
  - RN - identifies the type of the restriction
  - O - operation: insert, delete, update
  - T - name of the table on which the operation is executed
  - V - name of table of which the restriction depends
- propagate(PN,O,T,V) - where
  - PN - identifies the type of the propagation
  - O - operation: insert, delete, update
  - T - name of the table on which the operation is executed
  - V - name of the table affected by the propagation

The meaning of the various types of rules is explained below. Recall that they refer to delete, insert or update operations on tuples of a table T which reference or are referenced by tuples of a table V. Tuples of T and V will be simply denoted by A and B, respectively.

Restrict Rules

- for deletion

- rd1 - Forbids deletion of A if some B references A.
- rd2 - Forbids deletion of an A that is the only A to reference some B.
- rd3 - Forbids deletion of A if reference to B in A is not null.

- for insertion
  - ri1 - Forbids insertion of A if B referenced in A does not exist.
  - ri2 - Forbids insertion of A if neither the B referenced in A exists nor the reference to B is null.
  - ri3 - Forbids insertion of A if the values in specified columns of the B referencing A are null.
  - ri4 - Forbids insertion of A with non-null reference to B if some values in specified columns of A are null.
- for update
  - ru1 - Forbids update in A of reference to B if neither the newly referenced B exists nor the new reference to B is null.
  - ru2 - Forbids update in A of reference to B if some B would no longer be referenced by any A.
  - ru3 - Forbids update in A of reference to B into a null reference.
  - ru4 - Forbids update of A if new reference to B is not null but some new values in specified columns of A are null.

Propagate Rules

- for deletion
  - pd1 - Propagates deletion of A to delete all B referencing A.
  - pd2 - Propagates deletion of A to update to null the references to A in B.
  - pd3 - Propagates deletion of A to delete each B that is no longer referenced by any A.
- for insertion
  - pi1 - Propagates insertion of A to insertion of a B referencing A.
  - pi2 - Propagates insertion of A to update an indicated B to make it reference A.

- for update
  - pu1 - Propagates update to null in A of some of the fields in A referencing B to also update to null the other fields of the reference.
  - pu2 - Propagates update in A of reference to B to delete those B no longer referenced by any A.
  - pu3 - Propagates an operation to update to null in A of reference to B to convert the operation into the deletion of A.
  - pu4 - Propagates update to null in A of values in specified columns to delete the B referenced by A.

5.5. Tables and Restrict/Propagate Rules

Depending on the type of a table T and on the options taken by the designer during the dialogue at the conceptual design phase, different restrict and propagate rules are attached to T, as indicated below.

The restrict and propagate rules are generated by separate clauses of the "gen_rules" predicate, as explained below. We shall designate by T and V both relational tables and the corresponding entity-relationship objects. R will designate a relationship. A and B will be understood, respectively, as tuples of T and V.

- *Table T, where T is_a V* - T and V are either entity or extended-entity tables. Insertion of A is restricted to the existence of a B with the same key.
- *Table T, where V is_a T* - T and V are either entity or extended_entity tables. Deletion of A propagates to the deletion of the B with the same key.
- *Table T, where T is in a binary one-to-n R where V is the identifying participant* - T is an entity or extended-entity table and V is an extended-entity table. If R is total with respect to T, insertion of A propagates to the update of a B, chosen by the user, to make it reference A. The deletion of A may propagate to referencing Bs or be restricted by their existence; the case of propagation has two sub-cases: if R is total with respect to V, the propagation consists of deleting the Bs, otherwise the reference to A in the Bs is updated to null.
- *Table T, where T is the identifying participant in a binary one-to-n R* - T is an extended-entity table and V is an entity or extended-entity table. The deletion of A may be restricted if A contains a non-null reference to a B; the case of propagation is trivial, since the participation of A in R is registered in the A tuple itself and is therefore deleted when A is deleted.

  In this paragraph, we consider the cases where R is total with respect to T. The insertion of A is always restricted by the existence of the B referenced.

The deletion of the participation of A in R (by an update of R attributes in A) may either propagate to the deletion of A or be forbidden.

Now consider R not total with respect to T. The insertion of A is restricted to either the existence of the referenced B or to the reference to B being null. Among the R attributes in A, some may not be undefined, in which case the insertion of A will also be restricted to these values not being null; likewise, the update of such values is restricted to non-null values. If any R field in A is updated to null, then this update propagates to nullify all other R fields.

Next we consider the cases where R is total with respect to V. If the option is that the deletion of the last participation of some B in R is permitted, then the deletion of the last A recording such participation propagates to the deletion of the no longer referenced B, the same propagation resulting from an update of A that changes or nullifies such reference. If the option is not to permit that a B become unreferenced, then both the deletion of an A referencing B and an update of A to change or nullify a reference to B are restricted by the reference to B not being the last one.

In any case, if R is neither total with respect to T nor V, the update of references to B are restricted to the existence of B or to the reference being null.

- *Table T, where T is a participant in a general relationship R* - T is an entity or extended-entity table. V is a relationship table corresponding to R. The deletion of A may propagate to or be restricted by the existence of participations of A in R. If R is total with respect to T, insertion of A necessarily propagates to an insertion into V of a tuple referencing A, to be supplied by the user.

- *Table T, for general relationship R* - T is a relationship table for R and V is an entity or extended-entity table for some participant of R. In all cases, insertion of A is restricted by the existence of the referenced B. If R is total with respect to V, deletion of A may either propagate to the deletion of B or be restricted if A is the last participation of B in R.

- *Table T, if T has a multi-valued attribute* - T may be an entity, extended-entity or relationship table associated with a repeating attribute table V. In all cases, deletion of A propagates to the deletion of B whenever the key of A is part of the key of B. If T is an extended-entity and the multi-valued attribute is an attribute of an R registered in T, an update to nullify R fields of A propagates to deleting B.

- *Table T representing multi-valued attribute* - T is a repeating attribute table, associated with some entity, extended-entity or relationship table V. In all cases insertion of A is restricted by the existence of the associated B. If V is an extended-entity and the multi-valued attribute is an attribute of an R

registered in V, insertion of A is additionally restricted by a non-null reference in the R fields of B.

We now make more precise the correspondence between tables and rules by indicating for each table, according to the various cases, the type of restrict or propagate rule applicable.

A few abbreviations will be used here: "¬" for "not" , "prop" for "propagates", "restr" for "is restricted", "wrt" for "with respect to". The types of tables involved are also abbreviated: "Ex" is "extended entity", "EE" is "entity or extended entity", "Rel" is "relationship", "EER" is "entity, extended entity or relationship", "Rep" is "repeating attribute".

- EE table T, such that T is_a V - ri1
- EE table T, such that V is_a T - pd1
- EE table T associated by binary one-to-n R to Ex table V
  - R total wrt T - pi2
  - delete T prop to R and R total wrt V - pd1
  - delete T prop to R and R ¬total wrt V - pd2
  - delete T restr by R - rd1
- Ex table T associated by binary one-to-n R to EE table V
  - delete T restr by R - rd3
  - R total wrt T - ri1
  - R total wrt T and delete R prop to V - pu3
  - R total wrt T and delete R restr by T - ru3
  - R ¬total wrt T - ri2, ri4, ru4, pu1
  - R ¬total wrt T and R total wrt V and delete R prop to V - pd3, pu2
  - R ¬total wrt T and R total wrt V and delete R restr by V - rd2, ru2
  - in any case - ru1
- EE table T associated by general R to Rel table V
  - delete T prop to R - pd1
  - delete T restr by R - rd1
  - R total wrt T - pi1

- Rel table T for general R with associated EE table V
  - in all cases - ri1
  - R total wrt V and delete R prop V - pd3
  - R total wrt V and delete R restr to V - rd2
- EER table T with Rep table V
  - in all cases - pd1
  - multi-valued attribute of R in Ex table T - pu4
- Rep table T of EER table V
  - in all cases - ri1
  - multi-valued attribute of R in Ex table V - ri3

6. THE INTERFACE GENERATION PHASE

6.1 Outline of the 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.

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 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.

6.2. Commands

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. If the **D** parameter is not supplied, 'SAMPLE' is assumed. 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.

fin. to terminate a session.

6.3. The Executable Restrict and Propagate Rules

The restrict and propagate clauses generated by a call to **implement** include an additional parameter containing the code to be executed by the prototyping tool. Their form is then:

- restrict(RN,O,T,V,R) - where
  - RN - identifies the type of the restriction (as in section 4.3);
  - O - operation: insert, delete, update
  - T - name of the table on which the operation is executed
  - V - name of table of which the restriction depends
  - R - program to test the restriction
- propagate(PN,O,T,V,P) - where
  - PN identifies the type of the propagation (as in section 4.3);

- O - operation: insert, delete, update
- T - name of the table on which the operation is executed
- V - name of the table affected by the propagation
- P - program to execute the propagation

7. THE PROTOTYPING TOOL

7.1. Commands

If the application interface file **F** was created at the interface generation phase, a prototyping session may be initiated 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**, **interf** 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 main purpose of the commands is 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 "select" causes all selected tuples to be displayed as lists. The execution of SQL commands may be traced, which is especially useful to follow the chains of commands induced by the propagate rules. The commands of the prototyping tool are given below (SQL commands appearing as arguments must be enclosed in single quotes, and their arguments of type string must be enclosed in *pairs* of single quotes):

execute(S). displays all tuples obtained by executing the "select" command S.

execute(U). U is an SQL insert, delete or update command to be executed subject to the restrict and propagate rules.

trace_sql(STATUS). if STATUS has the value "on" (the default), a tracing mechanism is enabled; if it is "off", the mechanism is disabled. If tracing is enabled, the original insert, delete or update operation determined by an "execute" com-

mand is displayed, as well as the operations executed as consequences (to test conditions or triggered).

cr_table(T). displays the "create_table" command that was executed (at the third phase of a design session) to create table T.

cr_index(T). displays the "create_index" command that was executed (at the third phase of a design session) to create the index associated with table T that enforces the uniqueness of the keys of T.

cr_tables. applies "cr_table" for all tables belonging to the application.

cr_indexes. applies "cr_index" for all indexes belonging to the application.

fin. to terminate a session.

7.2. Errors

Error conditions detected by SQL are denoted by negative numbers. The interface issues special messages for the following errors (the message for the first error is indicated in "gen_tab_index" and all the others in "sql1", both of which are auxiliary predicates):

-601 attempt to create a table that already exists

-410 attempt to store a character string in a numeric field

-407 attempt to store a null in a field where nulls are not allowed

-803 attempt to create a duplicate tuple, i.e. a tuple with the same key value as an existing tuple (this is seen by SQL as a violation of an index with "unique" option)

The other SQL errors are simply indicated by a generic message, where the negative error condition number is displayed. Warnings from SQL are disregarded.

Errors corresponding to *restrict* and *propagate* clauses are also signalled by messages. The former occur when a pre-condition for an operation fails; the latter, when a triggered operation cannot be executed due, in turn, to some error. The error messages are indicated in template-rules whose predicate name is *message*.

7.3. Structure of the EXECUTE Predicate

The **execute** predicate for the "select" SQL command simply lists all tuples satisfying the query in a list format.

For "insert", "delete" and "update" SQL commands the execution is monitored by the application of the generated "restrict" and "propagate" rules. To this end, **execute** calls **execute1** and, in case of success, forces an SQL "commit", otherwise it issues a "rollback". In turn, **execute1** proceeds along the following 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" 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.

The analysis step (step 2) differs according to the type of operation. In each case, the "analyse" auxiliary predicate unifies its last argument, V, with the values involved. For "insert", V is the tuple to be inserted formatted as a list of values. For "delete", V is the list of tuples to be deleted, given the "where" clause; for this purpose, such tuples are fetched from the database by a "select" command. For "update", V is a list of pairs of tuples (T1,T2), where T1 is the current tuple and T2 the result of updating T1; again, to obtain the T1 tuples a "select" command is issued against the database.

8. 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 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.

Much remains to be done in at least two directions: refining the strategy and further developing the implementation, especially 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 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).

[FCT] A. L. Furtado, M. A. Casanova and L. Tucherman - A framework for design/redesign experts - Proc. of the First International Conference on Expert Database Systems - L. Kerschberg (ed.) - (1986) 313-328.

[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).

APPENDIX I - DESIGN SESSION IN THE NORMAL ENVIRONMENT

To initiate a design session in the normal CMS environment, enter the command:

chris

If in a previous session the log has been saved in a file F, enter

restore(F).

as the first command. No further call to this command should be made during the session.

The other commands mentioned in the windows environment to be issued from the 'EXEC' window are all usable: **entity**, **relationship**, **log**, **like**, **keep**, **drop** (with one argument) and **fin**. However, the effect of **log** is simply to enumerate the names of the predicates in the log. If you want to list one such predicate, say P, then enter

list(P).

During the dialogue, the commands **like**, **log**, **end**, **valid** and **drop** (without arguments) can be used. The behaviour of **log** is that described above.

Menus are also displayed, in some cases, to show what answers are possible. However, this is just to facilitate choosing the answer - you must enter the selected answer explicitly and in full.

For the logical design phase, there is no difference with respect to the windows environment. The **structure**, **behaviour** and **schema** commands have the same effect.

Finally, for the interface generation phase, there is again no difference with respect to the windows environment. The command **implement** has the same effect.

The rest of this appendix shows a complete session that specifies the conceptual schema corresponding to the entity relationship diagram in Figure 1.

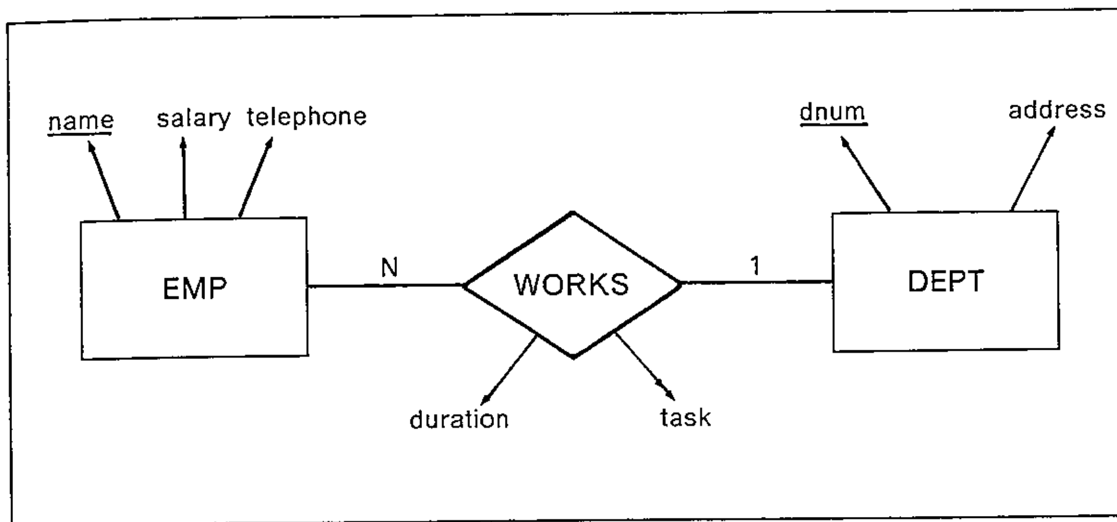


Figure 1 - Diagram of the Conceptual Schema

chris

--- VM/PROLOG 06/04/85 5785-ABH (C) Copyright IBM Corp. 1985 ---
61MS SUCCESS
<- initiate() .
0MS SUCCESS
<- pragma(lcomment,1) .
0MS SUCCESS
<- addax(command(drop,drop),1) .
3MS SUCCESS
<- clear() .
2MS SUCCESS
<- clear() .

entity(emp).

indicate attributes of emp

name.

on what type is name defined?
choose from the menu:
float
integer
char(V1)

char(20).

is attribute name repeating in emp?

no.

can attribute name be undefined in emp?

no.

indicate attributes of emp
= = > name

salary.

on what type is salary defined?
choose from the menu:
float
integer
char(V1)

integer.

is attribute salary repeating in emp?

no.

can attribute salary be undefined in emp?

no.

indicate attributes of emp
= = > name
= = > salary

telephone.

on what type is telephone defined?
choose from the menu:
float
integer
char(V1)

integer.

is attribute telephone repeating in emp?

no.

can attribute telephone be undefined in emp?

yes.

indicate attributes of emp
= = > name
= = > salary
= = > telephone

end.

which attributes identify emp?
choose from the menu:
name
salary

end.

entities need at least one identifying attribute

which attributes identify emp?
choose from the menu:
name
salary

name.

which attributes identify emp?
= = > name
choose from the menu:
salary

end.

99MS SUCCESS
<- entity(emp) .

entity(dept).

of what entities is dept a sub-class?
choose from the menu:
emp

end.

indicate attributes of dept

dnum.

on what type is dnum defined?
choose from the menu:
float
integer
char(V1)

integer.

is attribute dnum repeating in dept?

no.

can attribute dnum be undefined in dept?

no.

indicate attributes of dept
= = > dnum

address.

on what type is address defined?
choose from the menu:
float
integer
char(V1)

char(10).

is attribute address repeating in dept?

no.

can attribute address be undefined in dept?

no.

indicate attributes of dept
= = > dnum
= = > address

end.

which attributes identify dept?
choose from the menu:
address
dnum

dnum.

which attributes identify dept?
= = > dnum
choose from the menu:
address

end.

79MS SUCCESS
<- entity(dept) .

structure.

emp /entity/
name *key*
salary
telephone

dept /entity/
dnum *key*
address

7MS SUCCESS
<- structure() .

relationship(works).

what entities are in works?

emp.

must emp always participate in works?

yes.

can partipations of emp in works be suppressed to the last one?

yes.

can emp participating in works be deleted?

yes.

what entities are in works?
= = > emp

dept.

can dept participating in works be deleted?

yes.

what entities are in works?
= = > emp
= = > dept

end.

which participants identify works?
choose from the menu:
dept
emp

emp.

which participants identify works?
= = > emp
choose from the menu:
dept

end.

indicate attributes of works
= = > name
= = > dnum

duration.

on what type is duration defined?
choose from the menu:
float
integer
char(V1)

integer.

is attribute duration repeating in works?

no.

can attribute duration be undefined in works?

yes.

indicate attributes of works
= = > name
= = > dnum
= = > duration

task.

on what type is task defined?
choose from the menu:
float
integer
char(V1)
integer.

integer.

is attribute task repeating in works?

yes.

indicate attributes of works
= = > name
= = > dnum
= = > duration
= = > task

log.

attribute
can_be_undef
del_ent_prop
del_rel_prop
id_participant
identifier
is_a
is_entity

is_relat
is_total
participant
repeating
type

indicate attributes of works
==> name
==> dnum
==> duration
==> task

list(attribute).

an attribute of emp is name
an attribute of emp is salary
an attribute of emp is telephone
an attribute of dept is dnum
an attribute of dept is address
an attribute of works is name
an attribute of works is dnum
an attribute of works is duration
an attribute of works is task

indicate attributes of works
==> name
==> dnum
==> duration
==> task

end.

127MS SUCCESS
<- relationship(works) .

structure.

dept /entity/
  dnum *key*
  address

emp /extended_entity/
  name *key*
  salary
  telephone
  dnum

duration

works_task /repeating_attribute/
name *key*
task *key*

32MS SUCCESS
<- structure() .

behaviour.

Restrictions:
insert emp / dept exists
insert emp / check non-null attributes of works
update emp / check non-null attributes of works
insert works_task / emp exists
update emp / if referenced dept exists
insert works_task / non-null reference to dept in emp

Propagations:
delete dept -> delete emp
update emp -> delete emp without dept
delete emp -> delete works_task
update emp -> delete unreferenced works_task
58MS SUCCESS
<- behaviour() .

keep(alpha).

207MS SUCCESS
<- keep(alpha) .

implement(db1).

356MS SUCCESS
<- implement(db1) .

fin.

APPENDIX II - PROTOTYPING SESSION

interf db1

--- VM/PROLOG 06/04/85 5785-ABH (C) Copyright IBM Corp. 1985 ---
0MS SUCCESS
<- pragma(list,1) .
0MS SUCCESS
<- pragma(allgoal,1) .
0MS SUCCESS
<- pragma(goalprint,0) .
0MS SUCCESS
<- pragma(long,1) .
20MS SUCCESS
<- zeta() .

cr_table(dept).

'create table dept (dnum integer not null ,
address char(10)) in sample'

1MS SUCCESS
<- cr_table(dept) .

execute('insert into dept values(123,"gvte")').

'insert into dept values(123,"gvte")' .

55MS SUCCESS
<- execute('insert into dept values(123,"gvte")') .

execute('insert into dept values(123,"mb01")').

'insert into dept values(123,"mb01")' .

key violation

5MS FAIL

cr_index(dept).

'create unique index idept on dept (dnum)'

1MS SUCCESS
<- cr_index(dept) .

execute('insert into dept values(456,"mb01")').

'insert into dept values(456,"mb01")' .

5MS SUCCESS
 <- execute('insert into dept values(456,"mb01")') .

execute('select * from dept').

[123,'gvte']
[456,'mb01']

5MS SUCCESS
 <- execute('select * from dept') .

execute('select * from dept').

[123,'gvte']
[456,'mb01']

6MS SUCCESS
 <- execute('select * from dept') .

cr_table(emp).

'create table emp (name char(20) not null ,
   salary integer not null , telephone integer ,
   dnum integer not null , duration integer) in sample'

1MS SUCCESS
 <- cr_table(emp) .

execute('insert into emp values("maria",100,null,123,null)').

'select * from dept' .

'insert into emp values("maria",100,null,123,null)' .

15MS SUCCESS
 <- execute('insert into emp values("maria",100,null,123,null)') .

execute('insert into emp values("vitor",null,3764,123,12)').

'select * from dept' .

'insert into emp values(''vitor'',null,3764,123,12)' .

nulls not allowed

15MS FAIL

execute('insert into emp values(''vitor'',50,3764,123,12)').

'select * from dept' .

'insert into emp values(''vitor'',50,3764,123,12)' .

14MS SUCCESS
<- execute('insert into emp values(''vitor'',50,3764,123,12)') .

execute('insert into emp values(''raul'',50,2222,789,12)').

'select * from dept' .

ri1: entity does not exist
when executing insert on emp
with values:
['raul',50,2222,789,12] .

12MS FAIL

execute('insert into emp values(''raul'',50,2222,456,12)').

'select * from dept' .

'insert into emp values(''raul'',50,2222,456,12)' .

15MS SUCCESS
<- execute('insert into emp values(''raul'',50,2222,456,12)') .

execute('select * from emp').

['maria',100,null,123,null]
['raul',50,2222,456,12]
['vitor',50,3764,123,12]

7MS SUCCESS
<- execute('select * from emp') .

cr_tables.

'create table dept (dnum integer not null ,
address char(10)) in sample'

'create table emp (name char(20) not null ,
salary integer not null , telephone integer ,
dnum integer not null , duration integer) in sample'

'create table works_task (name char(20) not null ,
task integer not null) in sample'

3MS SUCCESS
<- cr_tables() .

execute('insert into works_task values("vitor",33)').

'select * from emp' .

'select * from emp' .

'insert into works_task values("vitor",33)' .

20MS SUCCESS
<- execute('insert into works_task values("vitor",33)') .

execute('insert into works_task values("vitor",44)').

'select * from emp' .

'select * from emp' .

'insert into works_task values("vitor",44)' .

21MS SUCCESS
<- execute('insert into works_task values("vitor",44)') .

execute('insert into works_task values("raul",33)').

'select * from emp' .

'select * from emp' .

'insert into works_task values("raul",33)' .

19MS SUCCESS
<- execute('insert into works_task values("raul",33)') .

execute('update emp set salary = null where name = ''maria''').

'update emp set salary = null where name = ''maria''' .

nulls not allowed

22MS FAIL

execute('delete from dept where dnum = 123').

'delete from dept where dnum = 123' .

'delete from emp where dnum = 123' .

'delete from works_task where name = ''maria''' .

'delete from works_task where name = ''vitor''' .

36MS SUCCESS
 <- execute('delete from dept where dnum = 123') .

execute('select * from emp').

['raul',50,2222,456,12]

5MS SUCCESS
 <- execute('select * from emp') .

execute('select * from dept').

[456,'mb01']

5MS SUCCESS
 <- execute('select * from dept') .

execute('select * from works_task').

['raul',33]

6MS SUCCESS
 <- execute('select * from works_task') .

fin.

APPENDIX III - HARDWARE AND SOFTWARE REQUIREMENTS

CHRIS requires a minimum of 3M bytes of main storage.

VM/Prolog, GDDM version 2.1 (or above), REXX and SQL/DS must be available. CHRIS uses the PGDDM, PREXX and PROSQL interfaces of VM/Prolog. While the latter is distributed with VM/Prolog, PGDDM and PREXX are "IBM internal use only" packages, obtainable from the AITOOLS repository (master in Paris, shadow in RIOVMSC and elsewhere). The three interfaces have to be *installed*, according to the instructions supplied in the respective documentation (the VM/Prolog manual for PROSQL, and the SCRIPT files in the PGDDM and PREXX packages).

CHRIS also uses the PROWIN and ETC packages, placed in the AITOOLS and in the GAVEA1 repositories. The GAVEA1 repository, located at RIOVMGVA, contains "loadlibs" and other files (package PRO_INTS), resulting from the installation of the three interfaces mentioned above. In RIOVMSC such files have been placed in the same disk where VM/Prolog resides. Users having access to these files do not have to worry about installing the interfaces. However, the interfaces may have to be re-installed if new versions of the respective software products or of the operating system are introduced.

To work in the windows environment, the terminal must have graphics capabilities that fully support the GDDM features, especially partitions. To work in the normal screen environment, GDDM and PGDDM are not needed.

The CHRIS package consists of the following files:

1. CHRIS SCRIPT
2. CHRIS EXEC
3. CHRIS PROLOG
4. INTERF EXEC
5. INTERF PROLOG
6. CHRISO EXEC
7. CHRISOBJ PROLOG01
8. INTERFO EXEC
9. INTERFO PROLOG01

File 1 contains the present document. Files 2-5 refer to the source version of CHRIS, and files 6-9 to the object version. The EXEC files initiate sessions. Files of type PROLOG are source, whereas PROLOG01 files are object.