SDT: A Database Schema Design and Translation Tool
Reference Manual
Draft 4.1

V.M. Markowitz and W. Fang

May 1991
SDT

A DATABASE SCHEMA DESIGN AND TRANSLATION TOOL*

Reference Manual

DRAFT 4.1

Victor M. Markowitz†
Weiping Fang‡

Data Management Group
Information and Computing Sciences Division
Lawrence Berkeley Laboratory
1 Cyclotron Road
Berkeley, CA 94720

May 1, 1991

Copyright © 1991 Lawrence Berkeley Laboratory

Do not redistribute without written permission from V. M. Markowitz† or A. Shoshani‡

† Author's E-mail address: V_Markowitz@lbl.gov Office phone number:(415) 486-6835
‡ Author's E-mail address: W_Fang@lbl.gov
§ E-mail address: A_Shoshani@lbl.gov Office phone number:(415) 486-5171
ABSTRACT

In this document we describe a database schema design and translation tool called SDT. SDT takes as input Extended Entity-Relationship (EER) schemas and generates relational database management (RDBMS) schemas. SDT consists of three main parts:

1. the first part maps EER schemas into abstract relational schemas,
2. the second part maps abstract relational schemas into schema definitions for RDBMSs; and
3. the third part generates the metadata regarding EER schemas, relational schemas, and their mappings.

SDT 4.1 targets SYBASE 4.0, INGRES 6.3, and INFORMIX 4.0.

NEW FEATURES Compared with SDT 3.1, SDT 4.1 has the following additional features:

1. a graphical editor for specifying and modifying EER schemas, called ERDRAW, can be used in conjunction with SDT 4.1. ERDRAW is described in technical report LBL-PUB-3084.

2. SDT 4.1 generates procedures for verifying the consistency of an existing database with regard to a set of referential integrity constraints associated with that database; such a verification is required when databases are loaded using RDBMS provided bulk copy utilities that bypass the referential integrity constraints, such as those of SYBASE 4.0 and INGRES 6.3.

3. Attributes and object-sets (entity-sets or relationship-sets) can be described using new description fields.

4. SDT 4.1 allows the specification of hierarchically organized subject terms for object-sets, attributes, and the association of subject terms with object-sets and attributes.

5. SDT 4.1 generates metadata describing EER schemas, relational schemas and their mappings; these metadata is embedded in appropriate insert operations ready for loading into a predefined metadata-base.

SDT was implemented using C, LEX, and YACC on Sun 3 and Sun 4 workstations under Sun Unix OS 4.0.3. and Sun Unix OS 4.1.

NOTE. This is a working, and therefore incomplete, document.
CONTENTS

I. Introduction ......................................................................................... 1

II Overview ....................................................................................... 2
  2.1 Outline of SDT ........................................................................... 2
  2.2 An Example ................................................................................ 2

III Input Formats .............................................................................. 3
  3.1 Input Format for EER Schemas .................................................. 3
  3.1 Input Format for Abstract Relational Schemas .............................. 6

IV Execution ....................................................................................... 8
  4.1 Command .................................................................................. 8
  4.2 Intermediary Output Files Generated by SDT for EER Input Schemas 8
    4.2.1 Abstract Relational Schema ................................................ 8
    4.2.2 Abstract Relational Schema after Merging ............................. 10
  4.3 Output Files Generated by SDT .................................................. 11
    4.3.1 SYBASE/SQL Schema ......................................................... 11
    4.3.2 INGRES/SQL Schema ......................................................... 18
    4.3.3 INFORMIX/SQL Schema .................................................... 24
    4.3.4 Referential Integrity Verification Procedures .......................... 25

V The Metadatabase ......................................................................... 30
  5.1 Schemas and Mappings ............................................................. 30
  5.2 Subject Terms ........................................................................... 30
  5.3 The SDT File for Metaschema Definition .................................... 33
  5.4 The SDT Metadata Output File ................................................. 35

VI The Program Structure of SDT .................................................. 42

References ......................................................................................... 45

A The Extended Entity-Relationship Model ................................. 46
  A.1 Fundamental Concepts ............................................................. 46
    A.1.1 Object-Sets ......................................................................... 46
    A.1.2 Value-Sets .......................................................................... 46
    A.1.3 Entity-Relationship Diagram .............................................. 47
    A.1.4 Entity-Identifier ................................................................. 47
    A.1.5 Existence Dependency ......................................................... 48
    A.1.6 Association and Involvement Cardinality ............................... 48
    A.1.7 Mandatory Involvement ...................................................... 49
    A.1.8 Role ................................................................................... 49
  A.2 Extended Concepts ................................................................. 50
    A.2.1 Generalization ................................................................. 50
    A.2.2 Types of Generalization .................................................... 50
    A.2.3 Extended Entity-Relationship Diagram .............................. 51
    A.2.4 Role Revisited ................................................................. 51
    A.2.5 Aggregation ...................................................................... 51
I. INTRODUCTION

We describe in this document a database schema design and translation tool (SDT) developed at Lawrence Berkeley Laboratory. The purpose of SDT is to provide a powerful and easy to use design interface for non-technical users, and to increase the productivity of the database design process. This entails insulating the schema designer from the underlying database management system (DBMS).

For the schema design interface we have chosen a version of the Extended Entity-Relationship (EER) model for the specification of the structure of information systems. The EER model we use includes, in addition to the basic construct of object (entity and relationship), both generalization and full aggregation abstraction capabilities. Once an EER schema is specified, SDT is employed in order to generate the corresponding relational DBMS (RDBMS) schema.

SDT consists of four main modules. The first SDT module takes EER schemas as input and generates abstract relational schemas. This module consists of three parts: the canonical mapping of EER schemas into normalized relational schemas; the assignment of names to relational attributes; and merging relations. The canonical mapping generates relational schemas, including key and referential integrity constraints. The high normal form (BCNF) of this schema ensures efficient update performance by the RDBMS. Name assignment can be customized in order to meet the needs of the user (e.g. short names, etc.). Finally, merging of relations reduces the number of relations, thus improving query performance.

The second SDT module takes abstract relational schemas as input and generates schema definitions for specific RDBMS, such as SYBASE, INGRES, and INFORMIX. For an RDBMS that supports the specification of triggers, such as SYBASE, or rules, such as INGRES, the main part of this module consists of generating the appropriate insert, delete, and update triggers or rules corresponding to the referential integrity constraints associated with the abstract relational schema.

The third SDT module generates procedures for verifying the consistency of a database with regard to a set of referential integrity constraints. Finally, the fourth SDT module generates metadata describing the EER and relational schemas and their mappings; these metadata is embedded in appropriate insert operations ready for loading into a predefined metadatabase.

Various research results related to the development of SDT are presented in references [3] to [9]. Most of the algorithms underlying SDT are described in [9].
II. OVERVIEW

2.1 Outline of SDT

Input: EER schema.

Output: SQL database definition for SYBASE 4.0, INGRES 6.3, or INFORMIX 4.0.

Steps:
1. Map the EER schema into an equivalent abstract relational schema.
   1.1 Check the correctness of the input EER schema; incorrect schemas are rejected.
   1.2 Map the EER schema into an abstract relational schema, with relations and relational attributes having symbolic (internal) names.
   1.3 Assign (externally meaningful) names to relations and relational attributes.
   1.4 Merge relations in the abstract relational schema.
2. Translate the abstract relational schema into database definition statements.
3. Generate procedures for verifying the consistency of a database with regard to the set of referential integrity constraints associated with the abstract relational schema associated with that database.
4. Generate the metadata information regarding the EER and relational schemas.

2.2 An Example

For illustration purposes, we use the EER schema represented in figure 2.1: PERSON, COURSE, and DEPARTMENT are independent entity-sets; FACULTY is a specialization of PERSON, OFFER is a relationship-set representing courses offered by departments, such that a course is offered by at most one department; and TEACH is a relationship-set representing the assignment of faculty members to teach courses offered by departments, such that a course is taught by at most one faculty member.

![Figure 2.1 An Extended Entity-Relationship Schema.](image)
III. INPUT FORMATS

3.1 Input Format for EER Schemas (Figure 3.1)

The syntax for specifying EER schemas is given in figure 3.1. A BNF-like notation is used in order to describe this syntax. Words in italic lower case letters denote non-terminals, while words in italic upper case letters and roman lower case letters denote terminals. Single-quoted characters are terminal delimiters whereas the rest are meta characters.

Notes:

1. A number must be in the syntax for a constant integer in C.
2. size is an upper bound on the number of objects in EER schema.
3. A domain must be in the form accepted by SYBASE/SQL, INGRES/SQL, or INFORMIX/SQL, respectively; the correct specification of the domain is the responsibility of the user.
4. An identifier is a letter or an underscore ('_'), possibly followed by a combined string of letters, underscores, and digits. Keywords are reserved identifiers.
5. The default for the null_rule when it is not specified, is NO NULLS.
6. For arc_type: ID, ISA, and ISA*, represent the arc types exactly as they appear in the EER schema; ONE represents a relationship cardinality of one and M represents a relationship cardinality of many; D1 represents both a relationship cardinality of one and mandatory involvement, and DM represents both a relationship cardinality of many and mandatory involvement.
specification ::= size object_subject_list
size ::= number
object_subject_list ::= object_subject | object_subject_list object_subject

object_subject ::= object | subject
  object ::= obj_head obj_tail ;
  obj_head ::= obj_name (' obj_type ')
  obj_name ::= identifier
  obj_type ::= entity | relationship
  entity ::= E | ENTITY
relationship ::= R | RELATIONSHIP
  obj_tail ::= attr_clause arc_clause descr_clause
attr_clause ::= ATTS ': attr_list | empty string
attr_list ::= attr | attr_list , attr
  attr ::= attr_name (' attr_type , descr , attr_subjects , domain null_rule ')
attr_name ::= identifier
  attr_type ::= ID | empty string
attr_subjects ::= subj_name | attr_subjects subj_name
  domain ::= data_type | data_type (' number ')
data_type ::= identifier
  null_rule ::= NO NULLS | NULLS ALLOWED | empty string
arc_clause ::= ARCS ': arc_list | empty string
arc_list ::= arc | arc_list , arc
  arc ::= obj_name (' arc_type , role ') | subj_name (' ST ')
arc_type ::= ID | ISA | ISA* | ONE | M | D 1 | DM
  role ::= identifier | empty string
descr_clause ::= DESCR ': descr
  descr ::= "" 'text' "
subject ::= subj_head subj_tail ;
subj_head ::= subj_name (' subj_type ')
subj_name ::= identifier
subject ::= SO | SA
subj_tail ::= broader_terms descr_clause
broader_terms ::= ARCS ': subject_list | empty string
  subject_list ::= subj_name (ISA,) | subject_list , subj_name (ISA,)
For example, the input file for the EER schema shown in figure 2.1, following the syntax given in figure 3.1, is given below:

```
8
PERSON(E)
  ATTRS: SSN(ID, "Social Security Number; Used as unique identifier.", , int NO NULLS),
          NAME(, "First and Last Name", , char(50) NULLS ALLOWED)
  DESCR: "";
FACULTY(E)
  ATTRS: RANK(, "Rank of faculty members", , char(25) NULLS ALLOWED)
  ARCS: PERSON(ISA, ), Course_Teaching(ST,)
  DESCR: "Faculty members";
DEPARTMENT(E)
  ATTRS: NAME(ID, "Name of Department", , char(30) NO NULLS)
  ARCS: Course_Offering(ST,)
  DESCR: "";
COURSE(E)
  ATTRS: NUMBER(ID, "Course number", , int NO NULLS)
  ARCS: Course_Offering(ST,)
  DESCR: "";
TEACH(R)
  ARCS: FACULTY(ONE, ), OFFER(M, ), Course_Teaching(ST,)
  DESCR: "Assignment of faculty members to teach offered courses";
OFFER(R)
  ARCS: DEPARTMENT(ONE, ), COURSE(M, ), Course_Offering(ST,), Course_Teaching(ST,)
  DESCR: "Offering of courses by departments";
Course_Offering (SO)
  DESCR: "";
Course_Teaching (SO)
  DESCR: "";
```
3.2 Input Format for Abstract Relational Schemas  (Figure 3.2)

The syntax of the language used for specifying input abstract relational schemas is given in figure 3.2. A BNF-like notation is used in order to describe this syntax. Non-terminals and terminals are denoted as

\[
\begin{align*}
\text{specification} & ::= \text{size relations} \\
\text{size} & ::= \text{number} \\
\text{relations} & ::= \text{relation} \mid \text{relations relation} \\
\text{relation} & ::= \text{RELATION} \text{relation name} \\
& \quad \left( \text{ attributes primary_key alternate_keys foreign_keys } \right) \\
\text{relation name} & ::= \text{identifier} \\
\text{attributes} & ::= \text{attribute} \mid \text{attributes attribute} \\
\text{attribute} & ::= \text{attribute name domain null_rule} \\
\text{attribute name} & ::= \text{identifier} \\
\text{domain} & ::= \text{data_type} \mid \text{data_type} \left( \text{ number } \right) \\
\text{data_type} & ::= \text{identifier} \\
\text{null_rule} & ::= \text{NO NULLS} \mid \text{NULLS ALLOWED} \mid \text{empty string} \\
\text{primary_key} & ::= \text{PRIMARY KEY} \left( \text{ attribute_names } \right) \\
\text{attribute_names} & ::= \text{attribute_name} \mid \text{attribute_names , attribute_name} \\
\text{alternate_keys} & ::= \text{empty string} \mid \text{alternate_keys alternate_key} \\
\text{alternate_key} & ::= \text{ALTERNATE KEY} \left( \text{ attribute_names } \right) \\
\text{foreign_keys} & ::= \text{empty string} \mid \text{foreign_keys foreign_key} \\
\text{foreign_key} & ::= \text{FOREIGN KEY} \left( \text{ attribute_names } \right) \\
& \quad \text{REFERENCES } \text{relation_name} \\
& \quad \text{INSERT option} \\
& \quad \text{DELETE option} \\
\text{option} & ::= \text{RESTRICTED}
\end{align*}
\]

Figure 3.2 Syntax for Abstract Relational Schemas.
above. Notes 1 to 5 above also apply for this definition.

For example, the abstract relational schema below follows the syntax given in figure 3.2:

3
RELATION DEPARTMENT (  
    NAME char(50) NO NULLS  
    PRIMARY KEY (NAME)  
)
RELATION COURSE (  
    NUMBER int NO NULLS  
    PRIMARY KEY (NUMBER)  
)
RELATION OFFER (  
    DEPARTMENT_NAME char(30) NO NULLS  
    COURSE_NUMBER int NO NULLS  
    PRIMARY KEY (COURSE_NUMBER)  
    FOREIGN KEY (DEPARTMENT_NAME)  
        REFERENCES DEPARTMENT  
            INSERT RESTRICTED  
            DELETE RESTRICTED  
    FOREIGN KEY (COURSE_NUMBER)  
        REFERENCES COURSE  
            INSERT RESTRICTED  
            DELETE RESTRICTED  
)
IV. EXECUTION

4.1 Command

\[ sdt \ [ -sT ] \ [ -cX ] \ [ -mY ] \ [ -tZ ] \ file \]

where

- \( T \) can be either \( e \) (for EER) or \( r \) (for relational), and specifies the type of input schema for SDT; parameters \( X \), \( Y \), and \( Z \) below are ignored when \( T = r \).
- If the \(-s\) option is not specified, EER schema is assumed by default.
- \( X \) can be either \( a \) (for association) or \( i \) (for involvement) and specifies the type of relationship cardinality used in the EER schema.
- If the \(-c\) option is not specified, association cardinality is assumed by default.
- \( Y \) can be either \( r \) (for restricted) or \( n \) (for no merging) and specifies the type of merging to be performed.
- If the \(-m\) option is not specified, the no merging is assumed by default.
- \( Z \) can be either \( s \) (for SYBASE), \( i \) (for INGRES 6.3), or \( x \) (for INFORMIX 4.0), and specifies the target RDBMS.
- If the \(-t\) option is not specified, SYBASE is assumed by default.
- \( file \) is the input file containing (1) an EER schema specification following the syntax given in Figure 3.1, or (2) an abstract relational schema specification following the syntax given in Figure 3.2.

4.2 Intermediary Output Files Generated by SDT for EER Input Schemas

4.2.1 Abstract Relational Schema

This file contains the abstract relational schema before merging. The file name consists of the name of the file containing the input EER schema, followed by "r".

For example, the abstract relational schema generated for the EER schema of figure 2.1, when no merging is requested, is given below. Note that names are assigned according to a Name Assignment algorithm selected by us, and which is designed to assign relations and relational attributes names as close as possible to the names of EER object-sets and attributes.
RELATION PERSON(
    SSN int NO NULLS
    NAME char (50) NULLS ALLOWED
    PRIMARY KEY (SSN)
)

RELATION FACULTY (
    SSN int NO NULLS
    RANK char (20) NO NULLS
    PRIMARY KEY (SSN)
    FOREIGN KEY (SSN)
        REFERENCES PERSON
            INSERT RESTRICTED
            DELETE RESTRICTED
)

RELATION DEPARTMENT(
    NAME char(30) NO NULLS
    PRIMARY KEY (NAME)
)

RELATION COURSE ( 
    NUMBER int NO NULLS
    PRIMARY KEY (NUMBER)
)

RELATION OFFER ( 
    DEPARTMENT_NAME char(30) NO NULLS
    COURSE_NUMBER int NO NULLS
    PRIMARY KEY (COURSE_NUMBER)
    FOREIGN KEY (DEPARTMENT_NAME)
        REFERENCES DEPARTMENT
            INSERT RESTRICTED
            DELETE RESTRICTED
    FOREIGN KEY (COURSE_NUMBER)
        REFERENCES COURSE
            INSERT RESTRICTED
            DELETE RESTRICTED
)

RELATION TEACH ( 
    FACULTY_SSN int NO NULLS
    COURSE_NUMBER int NO NULLS
    PRIMARY KEY (COURSE_NUMBER)
    FOREIGN KEY (FACULTY_SSN)
        REFERENCES FACULTY
            INSERT RESTRICTED
            DELETE RESTRICTED
    FOREIGN KEY (COURSE_NUMBER)
        REFERENCES OFFER
            INSERT RESTRICTED
            DELETE RESTRICTED
)
4.2.2 Abstract Relational Schema after Merging

This file contains the abstract relational schema after merging relations, if such a merging is requested. If merging is done at all, a file with name being the input EER schema, followed by ".m".

For example, the abstract relational schema generated for the EER schema of figure 2.1, if merging is requested, is given below. Note that SDT first finds the relations that can be merged and then performs their merging.

*Merged Relations*: course, offer, teach

```sql
RELATION PERSON (  
  SSN int NO NULLS  
  NAME char(50) NULLS ALLOWED  
  PRIMARY KEY (SSN)  
)  

RELATION FACULTY (  
  SSN int NO NULLS  
  RANK char(20) NO NULLS  
  PRIMARY KEY (SSN)  
  FOREIGN KEY (SSN)  
    REFERENCES PERSON  
      INSERT RESTRICTED  
      DELETE RESTRICTED  
)  

RELATION DEPARTMENT (  
  NAME char(30) NO NULLS  
  PRIMARY KEY (NAME)  
)  

RELATION COURSE(  
  FACULTY_SSNN int NULLS ALLOWED  
  DEPARTMENT_NAME char(30) NULLS ALLOWED  
  NUMBER int NO NULLS  
  PRIMARY KEY (NUMBER)  
  FOREIGN KEY (FACULTY_SSNN)  
    REFERENCES FACULTY  
      INSERT RESTRICTED  
      DELETE RESTRICTED  
  FOREIGN KEY (DEPARTMENT_NAME)  
    REFERENCES DEPARTMENT  
      INSERT RESTRICTED  
      DELETE RESTRICTED
)  
```
4.3 Output Files Generated by SDT

The database definition generated by SDT is contained in three files consisting of (1) the table (relation) definitions; (2) the index (key) definitions; and (3) the referential integrity constraints in declarative or procedural form. Two additional files contain (4) the procedures for verifying the referential integrity of an existing database, and (5) the metadata loading operations. The five file names containing the SDT output, consist of the name of the file containing the input EER schema, followed by (1) ".relations[Z]", (2) ".keys[Z]", (3) ".refint[Z]", (4) ".check[Z]", and (5) ".meta[Z]", respectively, where Z is either s (for SYBASE), i (for INGRES), or x (for INFORMIX).

Note: for INFORMIX only three files are currently generated, namely (1), (2), and (5).

The files generated by SDT can be loaded together. However, if data loading utilities provided by RDBMSs (such as the bcp utility of SYBASE) are going to be employed for loading data into the database, then for efficiency reasons it is preferable to load only the table definitions, then load the data into the database, and then load the index and referential integrity definitions (for more details consult the manuals of the RDBMS used).

4.3.1 SYBASE/SQL Schema

The SYBASE database definition is contained in three files consisting of (1) the table (relation) definitions; (2) the index (key) definitions; and (3) the trigger (referential integrity) procedures. The file names consist of the name of the file containing the input EER schema, followed by (1) ".relations.s", (2) ".keys.s", and (3) ".refint.s", respectively. The files are in ready-to-be-input-to-SYBASE form.

Examples for these files are given below.

The SYBASE schema definition corresponding to the merged abstract relational schema in section 4.2.2 above is given below:

File ExSybase_relations.s

    create database ExSybase
    go
    use ExSybase
    go
    create table PERSON (  
        SSN int not null,  
        NAME char(50) null  
    )
    create table FACULTY (  
        SSN int not null,  
        RANK char(25) null  
    )
create table DEPARTMENT (  
    NAME char(30) not null
  )
create table COURSE (  
    FACULTY_SSN int null,  
    DEPARTMENT_NAME char(30) null,  
    NUMBER int not null
  )
goquit

File ExSybase_keys.s :  
use ExSybase  
go  
cREATE unique clustered index indexPERSON on PERSON (SSN)  
cREATE unique clustered index indexFACULTY on FACULTY (SSN)  
cREATE unique clustered index indexDEPARTMENT on DEPARTMENT (NAME)  
cREATE unique clustered index indexCOURSE on COURSE (NUMBER)  
go  
sp_primarykey PERSON, SSN  
go  
sp_primarykey FACULTY, SSN  
go  
sp_primarykey DEPARTMENT, NAME  
go  
sp_primarykey COURSE, NUMBER  
go  
sp_foreignkey FACULTY, PERSON, SSN  
go  
sp_foreignkey COURSE, FACULTY, FACULTY_SSN  
go  
sp_foreignkey COURSE, DEPARTMENT, DEPARTMENT_NAME  
go  
quit

File ExSybase_refint.s :  
use ExSybase  
go  
CREATE trigger deletePERSON on PERSON  
for delete as  
begin  
  declare @delFACULTY int  
  select @delFACULTY = count(*) from deleted, FACULTY  
    where deleted.SSN = FACULTY.SSN  
  if @delFACULTY > 0  
    begin  
      raiserror 70002 "Cannot delete from PERSON because of"  
      print "existing reference from FACULTY"  
      select * from deleted  
        where exists
(select * from FACULTY
   where deleted.SSN = FACULTY.SSN)
   rollback transaction
end
end

create trigger updatePERSON on PERSON
for update as
begin
declare @row int, @delFACULTY int
select @row = @@rowcount
if update (SSN)
begin
   select @delFACULTY = count(*) from FACULTY
   where exists
      (select * from deleted
       where deleted.SSN = FACULTY.SSN)
   and not exists
      (select * from inserted
       where inserted.SSN = FACULTY.SSN)
   if 0 != @delFACULTY
   begin
      raiserror 70003 "Cannot update PERSON because of'
      print "existing reference from FACULTY"
      select * from deleted
      where exists
         (select * from FACULTY
          where deleted.SSN = FACULTY.SSN)
      and not exists
         (select * from inserted
          where inserted.SSN = FACULTY.SSN)
      rollback transaction
   end
   end
end

create trigger insertFACULTY on FACULTY
for insert as
begin
declare @row int, @insPERSON int, @nullPERSON int
select @row = @@rowcount
select @nullPERSON = 0
select @insPERSON = count(*) from inserted, PERSON
   where inserted.SSN = PERSON.SSN
if @nullPERSON + @insPERSON != 1 * @row
begin
   raiserror 70001 "Cannot insert into FACULTY because of"
   print "missing reference to PERSON"
   select * from inserted
   where not exists
      (select * from PERSON
      )
where inserted.SSN = PERSON.SSN)
rollback transaction
end
end

go
create trigger deleteFACULTY on FACULTY
for delete as
begin
  declare @delCOURSE int
  select @delCOURSE = count(*) from deleted, COURSE
  where deleted.SSN = COURSE.FACULTY_SSN
  if @delCOURSE > 0
  begin
    raiserror 70002 "Cannot delete from FACULTY because of"
    print "existing reference from COURSE"
    select * from deleted
    where exists
      (select * from COURSE
      where deleted.SSN = COURSE.FACULTY_SSN)
  rollback transaction
  end
end

go
create trigger updateFACULTY on FACULTY
for update as
begin
  declare @row int, @delCOURSE int, @insPERSON int, @nullPERSON int
  select @row = @@rowcount
  if update (SSN)
  begin
    select @delCOURSE = count (*) from COURSE
    where exists
      (select * from deleted
      where deleted.SSN = COURSE.FACULTY_SSN)
    and not exists
      (select * from inserted
      where inserted.SSN = COURSE.FACULTY_SSN)
    select @nullPERSON = 0
    select @insPERSON = count(*) from inserted, PERSON
    where inserted.SSN = PERSON.SSN
    if @nullPERSON + @insPERSON
    != 1 * @row + @delCOURSE
    begin
      raiserror 70003 "Cannot update FACULTY because of"
      if @delCOURSE != 0
      begin
        print "existing reference from COURSE"
        select * from deleted
        where exists
          (select * from COURSE
          where deleted.SSN = COURSE.FACULTY_SSN)
and not exists
   (select * from inserted
    where deleted.SSN = inserted.SSN)
end
if @nullPERSON + @insPERSON != @row
begin
   print "missing reference to PERSON"
   select * from inserted
    where not exists
       (select * from PERSON
        where inserted.SSN = PERSON.SSN)
end
rollback transaction
end
end
end
go
create trigger deleteDEPARTMENT on DEPARTMENT
for delete as
begin
declare @deICOURSE int
select @deICOURSE = count(*) from deleted, COURSE
   where deleted.NAME = COURSE.DEPARTMENT_NAME
if @deICOURSE > 0
begin
   raiserror 70002 "Cannot delete from DEPARTMENT because of"
   print "existing reference from COURSE"
   select * from deleted
    where exists
       (select * from COURSE
        where deleted.NAME = COURSE.DEPARTMENT_NAME)
   rollback transaction
end
end
go
create trigger updateDEPARTMENT on DEPARTMENT
for update as
begin
declare @row int, @delCOURSE int
select @row = @@rowcount
if update (NAME)
begin
   select @delCOURSE = count(*) from COURSE
      where exists
         (select * from deleted
          where deleted.NAME = COURSE.DEPARTMENT_NAME)
      and not exists
         (select * from inserted
          where inserted.NAME = COURSE.DEPARTMENT_NAME)
   if 0 != @delCOURSE
   begin

raiserror 70003 "Cannot update DEPARTMENT because of"
print "existing reference from COURSE"
select * from deleted
    where exists
        (select * from COURSE
            where deleted.NAME = COURSE.DEPARTMENT_NAME)
    and not exists
        (select * from inserted
            where deleted.NAME = inserted.NAME)
rollback transaction
end
end
end
go
create trigger insertCOURSE on COURSE
for insert as
begin
declare @row int, @insFACULTY int,
    @nullFACULTY int, @insDEPARTMENT int, @nullDEPARTMENT int
select @row = @@rowcount
select @nullFACULTY = count(*) from inserted
    where inserted.FACULTY_SSN = null
select @insFACULTY = count(*) from inserted, FACULTY
    where inserted.FACULTY_SSN = FACULTY.SSN
select @nullDEPARTMENT = count(*) from inserted
    where inserted.DEPARTMENT_NAME = null
select @insDEPARTMENT = count(*) from inserted, DEPARTMENT
    where inserted.DEPARTMENT_NAME = DEPARTMENT.NAME
if @nullFACULTY + @insFACULTY +
    @nullDEPARTMENT + @insDEPARTMENT != 2 * @row
    begin
        raiserror 70001 "Cannot insert into COURSE because of"
        if @nullFACULTY + @insFACULTY != @row
            begin
                print "missing reference to FACULTY"
                select * from inserted
                    where not exists
                        (select * from FACULTY
                            where inserted.FACULTY_SSN = FACULTY.SSN)
            end
        end
        if @nullDEPARTMENT + @insDEPARTMENT != @row
            begin
                print "missing reference to DEPARTMENT"
                select * from inserted
                    where not exists
                        (select * from DEPARTMENT
                            where inserted.DEPARTMENT_NAME=DEPARTMENT.NAME)
            end
        rollback transaction
    end
end
go
create trigger updateCOURSE on COURSE
for update as
begin
declare @row int, @insFACULTY int, @nullFACULTY int,
       @insDEPARTMENT int, @nullDEPARTMENT int
select @row = @@rowcount
if update (NUMBER) or
   update (FACULTY_SSN) or
   update (DEPARTMENT_NAME)
begin
   select @nullFACULTY = count(*) from inserted
   where inserted.FACULTY_SSN = null
   select @insFACULTY = count(*) from inserted, FACULTY
   where inserted.FACULTY_SSN = FACULTY.SSN
   select @nullDEPARTMENT = count(*) from inserted
   where inserted.DEPARTMENT_NAME = null
   select @insDEPARTMENT = count(*) from inserted, DEPARTMENT
   where inserted.DEPARTMENT_NAME = DEPARTMENT.NAME
if @nullFACULTY + @insFACULTY + @nullDEPARTMENT + @insDEPARTMENT
   != 2 * @row
begin
   raiserror 70003 "Cannot update COURSE because of"
   if @nullFACULTY + @insFACULTY != @row
   begin
       print "missing reference to FACULTY"
       select * from inserted
       where not exists
       (select * from FACULTY
        where inserted.FACULTY_SSN = FACULTY.SSN)
   end
   if @nullDEPARTMENT + @insDEPARTMENT != @row
   begin
       print "missing reference to DEPARTMENT"
       select * from inserted
       where not exists
       (select * from DEPARTMENT
        where inserted.DEPARTMENT_NAME=DEPARTMENT.NAME)
   end
   rollback transaction
end
end
end
go
quit
4.3.2 INGRES/SQL Schema

The INGRES database definition is contained in three files consisting (1) the table (relation) definitions; (2) the index (key) definitions; and (3) the trigger (referential integrity) procedures. The file names consist of the name of the file containing the input EER schema, followed by (1) "relations.i", (2) "keys.i", and (3) "refint.i", respectively. The files are in ready-to-be-input-to-INGRES form. Examples for these files are given below.

The INGRES schema definition corresponding to the merged abstract relational schema in section 4.2.2 above is given below:

File ExIngres_relations.i

```
CREATE TABLE PERSON (  
    SSN integer NOT NULL,  
    NAME char(50) WITH NULL
);
CREATE TABLE FACULTY (  
    SSN integer NOT NULL,  
    RANK char(25) WITH NULL
);
CREATE TABLE DEPARTMENT (  
    NAME char(30) NOT NULL
);
CREATE TABLE COURSE (  
    FACULTY_SSN integer WITH NULL,  
    DEPARTMENT_NAME char(30) WITH NULL,  
    NUMBER integer NOT NULL
);
\go
\quit
```

File ExIngres_keys.i:

```
CREATE UNIQUE INDEX idxPERSON on PERSON (SSN);
CREATE UNIQUE INDEX idxFACULTY on FACULTY (SSN);
CREATE UNIQUE INDEX idxDEPARTMENT on DEPARTMENT (NAME);
CREATE UNIQUE INDEX idxCOURSE on COURSE (NUMBER);
\go
\quit
```
CREATE PROCEDURE p_delPERSON (o_SSN integer, o_NAME char(50)) AS
DECLARE
    msg VARCHAR(256) NOT NULL; check_val INTEGER;
BEGIN
    SELECT COUNT(*) INTO :check_val FROM FACULTY
    WHERE SSN = :o_SSN;
    IF check_val > 0 THEN
        msg = 'Error 1: FACULTY '' + :o_SSN + ''' found.';
        RAISE ERROR 1 :msg;
        RETURN;
    ENDIF;
    msg = 'PERSON deleted''
         '(SSN = '' + :o_SSN + ''' , NAME = '' + :o_NAME + ''');
MESSAGE :msg;
END;
\go
CREATE RULE r_delPERSON AFTER DELETE FROM PERSON
EXECUTE PROCEDURE p_delPERSON
(o_SSN = old.SSN, o_NAME = old.NAME);
\go
CREATE PROCEDURE p_updPERSON (o_SSN integer, o_NAME char(50),
    n_SSN integer, n_NAME char(50)) AS
DECLARE
    msg VARCHAR(256) NOT NULL;
    check_val INTEGER;
BEGIN
    SELECT COUNT(*) INTO :check_val FROM FACULTY
    WHERE SSN = :o_SSN;
    IF check_val > 0 THEN
        msg = 'Error 1: FACULTY '' + :o_SSN + ''' found.';
        RAISE ERROR 1 :msg;
        RETURN;
    ENDIF;
    msg = 'PERSON updated''
         '(SSN = '' + :n_SSN + ''' , NAME = '' + :n_NAME + ''');
MESSAGE :msg;
END;
\go
CREATE RULE r_updPERSON AFTER UPDATE OF PERSON
EXECUTE PROCEDURE p_updPERSON
(o_SSN = old.SSN, o_NAME = old.NAME,
    n_SSN = new.SSN, n_NAME = new.NAME);
CREATE PROCEDURE p_insFACULTY (n_SSN integer, n_RANK char(25)) AS
DECLARE
  msg VARCHAR(256) NOT NULL;
  check_val INTEGER;
BEGIN
  IF n_SSN IS NOT NULL THEN
    SELECT COUNT(*) INTO :check_val FROM PERSON
    WHERE SSN = :n_SSN;
    IF check_val = 0 THEN
      msg = 'Error 1: PERSON "" + :n_SSN + "" not found.';
      RAISE ERROR 1 :msg;
    RETURN;
  ENDIF;
  ELSE
    msg = 'Error 2: FACULTY: nulls in SSN not allowed.';
    RAISE ERROR 2 :msg;
    RETURN;
  ENDIF;
  msg = 'FACULTY inserted' +
    'SSN = "" + :n_SSN + "", RANK = "" + :n_RANK + ");'
MESSAGE :msg;
END;
\go
CREATE RULE r_insFACULTY AFTER INSERT INTO FACULTY
EXECUTE PROCEDURE p_insFACULTY
  (n_SSN = new.SSN, n_RANK = new.RANK);
\go
CREATE PROCEDURE p_delFACULTY (o_SSN integer, o_RANK char(25)) AS
DECLARE
  msg VARCHAR(256) NOT NULL;
  check_val INTEGER;
BEGIN
  SELECT COUNT(*) INTO :check_val FROM COURSE
  WHERE FACULTY_SSNI = :o_SSN;
  IF check_val > 0 THEN
    msg = 'Error 1: COURSE "" + :o_SSN + "" found.';
    RAISE ERROR 1 :msg;
  RETURN;
  ENDIF;
  msg = 'FACULTY deleted'
    'SSN = "" + :o_SSN + ", RANK = "" + :o_RANK + ");'
MESSAGE :msg;
END;
\go
CREATE RULE r_delFACULTY AFTER DELETE FROM FACULTY
EXECUTE PROCEDURE p_delFACULTY
  (o_SSN = old.SSN, o_RANK = old.RANK);
\go
CREATE PROCEDURE p_updFACULTY (o_SSN integer, o_RANK char(25),
n_SSN integer, n_RANK char(25)) AS

DECLARE
    msg VARCHAR(256) NOT NULL;
    check_val INTEGER;
BEGIN
    SELECT COUNT(*) INTO :check_val FROM COURSE
    WHERE FACULTY_SSN = :o_SSN;
    IF check_val > 0 THEN
        msg = 'Error 1: COURSE ', + :o_SSN + ' found.';
        RAISE ERROR 1 :msg;
        RETURN;
    ENDIF;
    IF n_SSN IS NOT NULL THEN
        SELECT COUNT(*) INTO :check_val FROM PERSON
        WHERE SSN = :n_SSN;
        IF check_val = 0 THEN
            msg = 'Error 1: PERSON ', + :n_SSN + ' not found.';
            RAISE ERROR 1 :msg;
            RETURN;
        ENDIF;
    ELSE
        msg = 'Error 2: FACULTY: nulls in SSN not allowed.';
        RAISE ERROR 2 :msg;
        RETURN;
    ENDIF;
    msg = 'FACULTY updated' +
        '('SSN = ', + :n_SSN + ', RANK = ', + :n_RANK + ')';
    MESSAGE :msg;
END;
\go
CREATE RULE r_updFACULTY AFTER UPDATE OF FACULTY
EXECUTE PROCEDURE p_updFACULTY
(o_SSN = old.SSN, o_RANK = old.RANK,
n_SSN = new.SSN, n_RANK = new.RANK);
\go
CREATE PROCEDURE p_delDEPARTMENT (o_NAME char(30)) AS

DECLARE
    msg VARCHAR(256) NOT NULL;
    check_val INTEGER;
BEGIN
    SELECT COUNT(*) INTO :check_val FROM COURSE
    WHERE DEPARTMENT_NAME = :o_NAME;
    IF check_val > 0 THEN
        msg = 'Error 1: COURSE ', + :o_NAME + ' found.';
        RAISE ERROR 1 :msg;
        RETURN;
    ENDIF;
    msg = 'DEPARTMENT deleted' +
        '('NAME = ', + :o_NAME + ')';
    MESSAGE :msg;
END;
\go
CREATE RULE cdeIDEPARTMENT AFTER DELETE FROM DEPARTMENT
EXECUTE PROCEDURE p_delDEPARTMENT (o_NAME = old.NAME);

CREATE PROCEDURE p_updDEPARTMENT (o_NAME char(30),
n_NAME char(30)) AS
DECLARE
  msg VARCHAR(256) NOT NULL;
  check_val INTEGER;
BEGIN
  SELECT COUNT(*) INTO :check_val FROM COURSE
  WHERE DEPARTMENT_NAME = :o_NAME;
  IF check_val > 0 THEN
    msg = 'Error 1: COURSE "' + :o_NAME + '" found.';
    RAISE ERROR 1 :msg;
    RETURN;
  ENDIF;
  msg = 'DEPARTMENT updated' +
    '(NAME = "' + :n_NAME + '"');
MESSAGE :msg;
END;

CREATE RULE r_updDEPARTMENT AFTER UPDATE OF DEPARTMENT
EXECUTE PROCEDURE p_updDEPARTMENT
(o_NAME = old.NAME,
n_NAME = new.NAME);

CREATE PROCEDURE p_insCOURSE
(n_FACULTY_SSN integer, n_DEPARTMENT_NAME char(30), n_NUMBER integer) AS
DECLARE
  msg VARCHAR(256) NOT NULL;
  check_val INTEGER;
BEGIN
  IF n_FACULTY_SSN IS NOT NULL THEN
    SELECT COUNT(*) INTO :check_val FROM FACULTY
    WHERE SSN = :n_FACULTY_SSN;
    IF check_val = 0 THEN
      msg = 'Error 1: FACULTY '" + :n_FACULTY_SSN + '" not found.';
      RAISE ERROR 1 :msg;
      RETURN;
    ENDIF;
  ENDIF;
  IF n_DEPARTMENT_NAME IS NOT NULL THEN
    SELECT COUNT(*) INTO :check_val FROM DEPARTMENT
    WHERE NAME = :n_DEPARTMENT_NAME;
    IF check_val = 0 THEN
      msg = 'Error 2: DEPARTMENT '" + :n_DEPARTMENT_NAME + '" not found.';
      RAISE ERROR 2 :msg;
    ENDIF;
  ENDIF;
END;
RETURN;
ENDIF;

msg = 'COURSE inserted' +
'(FACULTY_SSN = '' + :n_FACULTY_SSN + ''' , DEPARTMENT_NAME = ''
+ :n_DEPARTMENT_NAME + ''' , NUMBER = '' + :n_NUMBER + '')';

MESSAGE :msg;
END;

CREATE RULE r_insCOURSE AFTER INSERT INTO COURSE
EXECUTE PROCEDURE p_insCOURSE (n_FACULTY_SSN = new.FACULTY_SSN,
 n_DEPARTMENT_NAME = new.DEPARTMENT_NAME, n_NUMBER = new.NUMBER)

CREATE PROCEDURE p_updCOURSE
(o_FACULTY_SSN integer, o_DEPARTMENT_NAME char(30), o_NUMBER integer,
 n_FACULTY_SSN integer, n_DEPARTMENT_NAME char(30), n_NUMBER integer) AS

DECLARE
 msg VARCHAR(256) NOT NULL;
 check_val INTEGER;
BEGIN
 IF n_FACULTY_SSN IS NOT NULL THEN
  SELECT COUNT(*) INTO :check_val FROM FACULTY
  WHERE SSN = :n_FACULTY_SSN;
  IF check_val = 0 THEN
   msg = 'Error 1: FACULTY'' + :n_FACULTY_SSN + ''' not found.';
   RAISE ERROR 1 :msg;
   RETURN;
  ENDIF;
 ENDIF;
 IF n_DEPARTMENT_NAME IS NOT NULL THEN
  SELECT COUNT(*) INTO :check_val FROM DEPARTMENT
  WHERE NAME = :n_DEPARTMENT_NAME;
  IF check_val = 0 THEN
   msg = 'Error 1: DEPARTMENT'' + :n_DEPARTMENT_NAME + ''' not found.';
   RAISE ERROR 1 :msg;
   RETURN;
  ENDIF;
 ENDIF;
 msg = 'COURSE updated' +
'(FACULTY_SSN = '' + :n_FACULTY_SSN + ''' , DEPARTMENT_NAME = ''
+ :n_DEPARTMENT_NAME + ''' , NUMBER = '' + :n_NUMBER + ''');

MESSAGE :msg;
END;

CREATE RULE r_updCOURSE AFTER UPDATE OF COURSE
EXECUTE PROCEDURE p_updCOURSE (o_FACULTY_SSN = old.FACULTY_SSN,
 o_DEPARTMENT_NAME = old.DEPARTMENT_NAME,
 o_NUMBER = old.NUMBER, n_FACULTY_SSN = new.FACULTY_SSN,
 n_DEPARTMENT_NAME = new.DEPARTMENT_NAME, n_NUMBER = new.NUMBER);
4.3.3 INFORMIX/SQL Schema

The INFORMIX database definition is contained in two files consisting of (1) the table (relation) definitions; and (2) the index (key) definitions. An additional file contains the (3) metadata loading operations. The file names consist of the name of the file containing the input EER schema, followed by (1) "relations.i", (2) "keys.i", and (3) "meta.x", respectively. The files are in ready-to-be-input-to-INFORMIX form. Examples of the files containing the table and index definitions are given below.

The INFORMIX schema definition corresponding to the merged abstract relational schema in section 4.2.2 above is given below:

File _ExInformix_relations.i_

```sql
create database ExInformix;
database ExInformix;

create table PERSON ( 
    SSN int not null, 
    NAME char(50)
);
create table FACULTY ( 
    SSN int not null, 
    RANK char(25)
);
create table DEPARTMENT ( 
    NAME char(30) not null
);
create table COURSE ( 
    FACULTY_SSN int, 
    DEPARTMENT_NAME char(30), 
    NUMBER int not null
);
close ExInformix;
```

File _ExInformix_keys.x_

```sql
database ExInformix;
create unique cluster index indexPERSON on PERSON (SSN);
create unique cluster index indexFACULTY on FACULTY (SSN);
create unique cluster index indexDEPARTMENT on DEPARTMENT (NAME);
close ExInformix;
```
4.3.4 Referential Integrity Verification Procedures

*SDT* generates procedures for verifying the referential integrity of an existing database. For every relation (table) in a database, a procedure for verifying the integrity of data in that relation is generated; the name of the procedure is of the form check_[T] where *T* is the name of the corresponding relation. For verifying the integrity of an entire database, a global procedure called check_all is provided.

4.3.4.1 Verification Procedures for SYBASE

The procedures for verifying the referential integrity of an existing SYBASE database defined as in section 4.3.1 above are given below:

File *ExSybase_check.s* :

```sql
use ExSybase
go
create procedure check_FACULTY as
begin
  if (select count(*) from PERSON) = 0
  begin
    select * into #1 from FACULTY
    if (select count(*) from #1) != 0
    begin
      print "The following tuples in FACULTY do not have references in PERSON"
      select * from #1
    end
  end
else
begin
  select * into #2 from FACULTY
  where not exists (select * from PERSON
  where FACULTY.SSN = PERSON.SSN)
  if (select count(*) from #2) != 0
  begin
    print "The following tuples in FACULTY do not have references in PERSON"
    select * from #2
  end
end
end
go
create procedure check_COURSE as
begin
  if (select count(*) from FACULTY) = 0
  begin
    select * into #1 from COURSE
    where COURSE.FACULTY_SSN is not null
    if (select count(*) from #1) != 0
    begin
      print "The following tuples in COURSE do not have references in FACULTY"
    end
  end
end
```
select * from #1
end
end
else
begin
select * into #2 from COURSE
where COURSE.FACULTY_SSN is not null
and not exists (select * from FACULTY
where COURSE.FACULTY_SSN = FACULTY.SSN)
if (select count(*) from #2) != 0
begin
print "The following tuples in COURSE do not have references in FACULTY"
select * from #2
end
end
if (select count(*) from DEPARTMENT) = 0
begin
select * into #3 from COURSE
where COURSE.DEPARTMENT_NAME is not null
if (select count(*) from #3) != 0
begin
print "The following tuples in COURSE do not have references in DEPARTMENT"
select * from #3
end
else
begin
select * into #4 from COURSE
where COURSE.DEPARTMENT_NAME is not null
and not exists (select * from DEPARTMENT
where COURSE.DEPARTMENT_NAME = DEPARTMENT.NAME)
if (select count(*) from #4) != 0
begin
print "The following tuples in COURSE do not have references in DEPARTMENT"
select * from #4
end
end
go
create procedure check_all as
begin
exec check_FACULTY
exec check_COURSE
end
go
quit
4.3.4.2 Verification Procedures for INGRES

The procedures for verifying the referential integrity of an existing INGRES database defined as in section 4.3.2 above are given below:

File ExIngres_check.i:

```sql
CREATE PROCEDURE check_FACULTY AS
    DECLARE
        msg VARCHAR(256) NOT NULL;
    BEGIN
        IF (SELECT COUNT(*) FROM PERSON) = 0
            BEGIN
                SELECT * INTO xxxx FROM FACULTY;
                IF (SELECT COUNT(*) FROM xxxx) != 0
                    BEGIN
                        msg = 'The following tuples in FACULTY' +
                             'do not have references in PERSON';
                        MESSAGE :msg;
                        SELECT * FROM xxxx;
                        DROP TABLE xxxx;
                    END;
            END
        ELSE
            BEGIN
                SELECT * INTO xxxx FROM FACULTY
                WHERE NOT EXISTS (SELECT * FROM PERSON
                                  WHERE FACULTY.SSN = PERSON.SSN);
                IF (SELECT COUNT(*) FROM xxxx) != 0
                    BEGIN
                        msg = 'The following tuples in FACULTY' +
                             'do not have references in PERSON';
                        MESSAGE :msg;
                        SELECT * FROM xxxx;
                        DROP TABLE xxxx;
                    END;
            END;
    END;
END;

\go
CREATE PROCEDURE check_COURSE AS
    DECLARE
        msg VARCHAR(256) NOT NULL;
    BEGIN
        IF (SELECT COUNT(*) FROM FACULTY) = 0
            BEGIN
                SELECT * INTO xxxx FROM COURSE
                WHERE COURSE.FACULTY_SSN IS NOT NULL;
                IF (SELECT COUNT(*) FROM xxxx) != 0
                    BEGIN
                        msg = 'The following tuples in COURSE' +
                             'do not have references in FACULTY';
                        MESSAGE :msg;
                        SELECT * FROM xxxx;
                        DROP TABLE xxxx;
                    END;
            END
        END;
```
'do not have references in FACULTY';
MESSAGE :msg;
SELECT * FROM xxxx;
DROP TABLE xxxx;
END;
ELSE
BEGIN
SELECT * INTO xxxx FROM COURSE
WHERE COURSE.FACULTY_SSN IS NOT NULL
AND NOT EXISTS (SELECT * FROM FACULTY
    WHERE COURSE.FACULTY_SSN = FACULTY.SSN);
IF (SELECT COUNT(*) FROM xxxx) != 0
BEGIN
    msg = 'The following tuples in COURSE' +
        'do not have references in FACULTY';
    MESSAGE :msg;
    SELECT * FROM xxxx;
    DROP TABLE xxxx;
END;
END;
ELSE
BEGIN
SELECT * INTO xxxx FROM COURSE
WHERE COURSE.DEPARTMENT_NAME IS NOT NULL;
IF (SELECT COUNT(*) FROM xxxx) != 0
BEGIN
    msg = 'The following tuples in COURSE' +
        'do not have references in DEPARTMENT';
    MESSAGE :msg;
    SELECT * FROM xxxx;
    DROP TABLE xxxx;
END;
END;
ELSE
BEGIN
SELECT * INTO xxxx FROM COURSE
WHERE COURSE.DEPARTMENT_NAME IS NOT NULL
AND NOT EXISTS (SELECT * FROM DEPARTMENT
    WHERE COURSE.DEPARTMENT_NAME = DEPARTMENT.NAME);
IF (SELECT COUNT(*) FROM xxxx) != 0
BEGIN
    msg = 'The following tuples in COURSE' +
        'do not have references in DEPARTMENT';
    MESSAGE :msg;
    SELECT * FROM xxxx;
    DROP TABLE xxxx;
END;
END;
END;
CREATE PROCEDURE check_all AS
  BEGIN
    EXECUTE PROCEDURE check_FACULTY;
    EXECUTE PROCEDURE check_COURSE;
  END;
\go
\quit
V. THE METADATABASE

The metadatabase schema consists of four main parts regarding: (1) the description of EER schemas; (2) the description of relational schemas; (3) the mapping of EER into relational schemas; and (4) subject term structures and associations.

5.1. Schemas and Mappings.

The metadatabase schema part regarding the description of EER schemas, the description of relational schemas, and the mapping of EER into relational schemas is shown in figure 5.1 and is self-explanatory.

5.2. Subject Terms.

Let $DB$ denote a relational database associated with an EER schema. Object-sets, attributes, and object instances represented in $DB$ can be associated with subject terms as follows:

1. The subject terms related to object-sets are grouped into an object-set called OBJECT-SET SUBJECT TERMS represented in the metadatabase associated with $DB$, as shown in figure 5.2. The subject terms related to attributes are grouped into an object-set called ATTRIBUTE SUBJECT TERMS represented in the metadatabase associated with $DB$, as shown in figure 5.2.

2. An object-set or attribute represented in $DB$, can be associated with one or several subject terms from OBJECT-SET SUBJECT TERMS or ATTRIBUTE SUBJECT TERMS, respectively. These associations are specified during the process of defining the schema for $DB$, and are stored in the metadatabase as instances of the relationship-sets OBJECT SUBJECT ASSOCIATION and ATTRIBUTE SUBJECT ASSOCIATION, respectively.

3. Subject terms of both object-set OBJECT-SET SUBJECT TERMS and object-set ATTRIBUTE SUBJECT TERMS can be organized in a classification hierarchy by associating every subject term with its broader and narrower terms. These classifications are represented as instances of relationship-sets OBJECT SUBJECT CLASSIFICATION and ATTRIBUTE SUBJECT CLASSIFICATION, respectively, as shown in figure 5.2.

4. Object-set and attribute subject terms are grouped together into an object-set called GLOBAL SUBJECT TERMS as shown in figure 5.2. Global subject terms can also be organized in a classification hierarchy by associating every global subject term with its broader and narrower terms, where these classifications are represented as instances of relationship-set GLOBAL SUBJECT CLASSIFICATION (see figure 5.2).
Figure 5.1. The Metadatabase: Information on the EER and Relational Schemas, and their Mapping.
5. The instances of an entity-set $E$ represented in $DB$ can be associated with subject terms by grouping these instances into specialization entity-sets of $E$. Accordingly, the classification of subject terms associated with entity-set instances is represented by relationships of the relationship-set OBJECT-SET CONNECTIONS of the metadatabase associated with $DB$ (see figure 5.2), where attribute TYPE is equal to ISA.

Subject terms are used for grouping related object-sets or attributes together by subject term, and are employed for schema browsing.

Figure 5.2. The Metadatabase: Subject Terms and Classifications.
5.3. The SDT File for Metaschema Definition.

The input file for SDT containing the (fixed) definition of the metadatabase schema is provided as part of the SDT package. The only part of this definition that must be adapted to the underlying DBMS are the attribute datatypes. For example, the SDT input file containing the schema definition of a metadatabase intended for SYBASE is given below:

```
28
RELATION_KEYS(E)
  ATTRS: Number(ID, "", int NO NULLS), Type("", char(30) NO NULLS)
  ARCS: RELATION_SCHEME(ID, )
  DESCR: ""
HAS(R)
  ARCS: OBJECT_ATTRIBUTE(DM, ), VALUE_SET(ONE, )
  DESCR: ""
OBJECT_SET(E)
  ATTRS: NAME(ID, "", char(30) NO NULLS), Description("", varchar(255) NULLS ALLOWED)
  DESCR: "Entity and Relationship Sets."
RELATION_SCHEME(E)
  ATTRS: NAME(ID, "", char(30) NO NULLS)
  DESCR: ""
REFERENTIAL_INTEGRITY(R)
  ATTRS: Insert_Rule("", char(30) NULLS ALLOWED), Delete_Rule("", char(30) NULLS ALLOWED),
         Update_Rule("", char(30) NULLS ALLOWED)
  ARCS: RELATION_SCHEME(M, TO), RELATION_KEYS(M, FROM)
  DESCR: ""
OBJECT_CONNECTION(R)
  ATTRS: Role(ID, "", char(30) NO NULLS), Connection_Type("", char(30) NO NULLS),
         Existence_Type("", char(30) NULLS ALLOWED)
  ARCS: OBJECT_SET(M, FROM), OBJECT_SET(M, TO)
  DESCR: ""
CONNECTION_MAPPING(R)
  ARCS: OBJECT_CONNECTION(ONE, ), REFERENTIAL_INTEGRITY(D1, )
  DESCR: ""
VALUE_SET(E)
  ATTRS: Name(ID, "", char(30) NO NULLS), Description("", varchar(255) NULLS ALLOWED)
  DESCR: ""
OBJECT_SET_MAPPING(R)
  ARCS: OBJECT_SET(M, ), RELATION_SCHEME(D1, )
  DESCR: ""
```
OBJECT_ATTRIBUTE(E)

ATTRS: NAME(ID, "", , char(30) NO NULLS), Null_Rule(, ",", , char(30) NULLS ALLOWED),
    Description(, "", , varchar(255) NULLS ALLOWED)
ARCS: OBJECT_SET(ID, )
DESCR: "Attributes for Entity and Relationship Sets."

RELATIONAL_ATTRIBUTE(E)

ATTRS: NAME(ID, "", , char(30) NO NULLS), Null_Rule(, ",", , char(30) NULLS ALLOWED)
ARCS: RELATION_SCHEME(ID, )
DESCR: "";

DOMAIN(E)

ATTRS: NAME(ID, "", , char(30) NO NULLS), Description(, "", , char(120) NULLS ALLOWED)
DESCR: "";

WITH_A(R)

ARCS: RELATIONAL_ATTRIBUTE(DM, ), DOMAIN(ONE, )
DESCR: "";

ATTRIBUTE_MAPPING(R)

ARCS: OBJECT_ATTRIBUTE(ONE, ), RELATIONAL_ATTRIBUTE(DM, )
DESCR: "";

VALUE_SET_MAPPING(R)

ARCS: VALUE_SET(ONE, ), DOMAIN(D1, )
DESCR: "";

KEY_ATTRIBUTE(R)

ARCS: RELATIONAL_ATTRIBUTE(M, ), RELATION_KEYS(DM, IN)
DESCR: "";

WITH_FORMAT(R)

ARCS: VALUE_SET(M, ), VALUE_FORMAT(M, )
DESCR: "";

CONSISTS_OF(R)

ARCS: VALUE_SET(M, ), VALUE_RANGES(M, )
DESCR: "";

VALUE_FORMAT(E)

ATTRS: Code(ID, "", , char(30) NO NULLS), Description(, "", , varchar(255) NO NULLS)
DESCR: "Format of Value Set."

VALUE_RANGES(E)

ATTRS: Code(ID, "", , char(30) NO NULLS), Upper_Bound(, "", , varchar(255) NULLS ALLOWED),
    Lower_Bound(, "", , varchar(255) NULLS ALLOWED),
    Description(, "", , varchar(255) NULLS ALLOWED)
DESCR: "Ranges of Value Set."

GLOBAL_SUBJ_TERM(E)

ATTRS: NAME(ID, "", , char(30) NO NULLS)
DESCR: "";
5.4. The SDT Metadata Output File.

As mentioned in the previous section, SDT generates a file containing metadata embedded in insertion operations appropriate for the underlying DBMS. The name of this file is ".meta.s" for SYBASE, ".meta.i" for INGRES, and ".meta.x" for INFORMIX, respectively.

For example, the metadata file corresponding to the EER schema described in section 2, and the abstract relational schema described in section 4.2.2, for a SYBASE metadatabase is given below:

File  ExSybase.meta.s

```sql
use Meta_ExSybase
go
insert OBJECT_SET(NAME, Description)
values("PERSON", "")
go
insert OBJECT_SET(NAME, Description)
values("FACULTY", "Faculty members")
go
insert OBJECT_SET(NAME, Description)
values("DEPARTMENT", "")
go
```
insert OBJECT_SET(NAME, Description)
values("COURSE", "")
go
insert OBJECT_SET(NAME, Description)
values("TEACH", "Represents assignments of faculty members to offered courses")
go
insert OBJECT_SET(NAME, Description)
values("OFFER", "Represents offering of courses by departments")
go
insert OBJECT_ATTRIBUTE(OBJECT_SET_NAME, NAME, Null_Rule, Description)
values("PERSON", "SSN", "NO NULLS", "Social Security Number; Used as unique identifier.")

insert OBJECT_ATTRIBUTE(OBJECT_SET_NAME, NAME, Null_Rule, Description)
values("PERSON", "NAME", "NULLS ALLOWED", "First and Last Name")

insert OBJECT_ATTRIBUTE(OBJECT_SET_NAME, NAME, Null_Rule, Description)
values("FACULTY", "RANK", "NULLS ALLOWED", "Rank of faculty members")

insert OBJECT_ATTRIBUTE(OBJECT_SET_NAME, NAME, Null_Rule, Description)
values("DEPARTMENT", "NAME", "NO NULLS", "Name of Department")

insert OBJECT_ATTRIBUTE(OBJECT_SET_NAME, NAME, Null_Rule, Description)
values("COURSE", "NUMBER", "NO NULLS", "Course number")

insert OBJECT_CONNECTION(FROM_OBJECT_SET_NAME, TO_OBJECT_SET_NAME, Role, Connection_Type, Existence_Type)
values("FACULTY", "PERSON", "!NONE!", "ISA", null)
go
insert OBJECT_CONNECTION(FROM_OBJECT_SET_NAME, TO_OBJECT_SET_NAME, Role, Connection_Type, Existence_Type)
values("TEACH", "OFFER", "!NONE!", "REL", null)
go
insert OBJECT_CONNECTION(FROM_OBJECT_SET_NAME, TO_OBJECT_SET_NAME, Role, Connection_Type, Existence_Type)
values("TEACH", "FACULTY", "!NONE!", "REL", null)
go
insert OBJECT_CONNECTION(FROM_OBJECT_SET_NAME, TO_OBJECT_SET_NAME, Role, Connection_Type, Existence_Type)
values("OFFER", "COURSE", "!NONE!", "REL", null)
go
insert OBJECT_CONNECTION(FROM_OBJECT_SET_NAME, TO_OBJECT_SET_NAME, Role, Connection_Type, Existence_Type)
values("OFFER", "DEPARTMENT", "!NONE!", "REL", null)
go
insert RELATION_SCHEME(NAME)
values("PERSON")

insert RELATION_SCHEME(NAME)
values("FACULTY")

insert RELATION_SCHEME(NAME)
values("DEPARTMENT")

go
insert RELATION_SCHEME(NAME)
    values("COURSE")

go
insert RELATION_SCHEME(NAME)
    values("TEACH")

go
insert RELATION_SCHEME(NAME)
    values("OFFER")

go
insert RELATIONAL_ATTRIBUTE(RELATION_SCHEME_NAME, NAME, Null_Rule)
    values("PERSON", "SSN", "not null")

go
insert RELATIONAL_ATTRIBUTE(RELATION_SCHEME_NAME, NAME, Null_Rule)
    values("PERSON", "NAME", "null")

go
insert RELATIONAL_ATTRIBUTE(RELATION_SCHEME_NAME, NAME, Null_Rule)
    values("FACULTY", "SSN", "not null")

go
insert RELATIONAL_ATTRIBUTE(RELATION_SCHEME_NAME, NAME, Null_Rule)
    values("FACULTY", "RANK", "null")

go
insert RELATIONAL_ATTRIBUTE(RELATION_SCHEME_NAME, NAME, Null_Rule)
    values("DEPARTMENT", "NAME", "not null")

go
insert RELATIONAL_ATTRIBUTE(RELATION_SCHEME_NAME, NAME, Null_Rule)
    values("COURSE", "FACULTY_SSN", "null")

go
insert RELATIONAL_ATTRIBUTE(RELATION_SCHEME_NAME, NAME, Null_Rule)
    values("COURSE", "DEPARTMENT_NAME", "null")

go
insert RELATIONAL_ATTRIBUTE(RELATION_SCHEME_NAME, NAME, Null_Rule)
    values("COURSE", "NUMBER", "not null")

go
insert OBJECT_SET_MAPPINGS(OBJECT_SET_NAME, RELATION_SCHEME_NAME)
    values("PERSON", "PERSON")

go
insert OBJECT_SET_MAPPINGS(OBJECT_SET_NAME, RELATION_SCHEME_NAME)
    values("FACULTY", "FACULTY")

go
insert OBJECT_SET_MAPPINGS(OBJECT_SET_NAME, RELATION_SCHEME_NAME)
    values("DEPARTMENT", "DEPARTMENT")

go
insert OBJECT_SET_MAPPINGS(OBJECT_SET_NAME, RELATION_SCHEME_NAME)
    values("COURSE", "COURSE")

go
insert OBJECT_SET_MAPPINGS(OBJECT_SET_NAME, RELATION_SCHEME_NAME)
    values("TEACH", "TEACH")

go
insert OBJECT_SET_MAPPINGS(OBJECT_SET_NAME, RELATION_SCHEME_NAME)
values("OFFER", "OFFER")
go
insert ATTRIBUTE_MAPPING(OBJECT_SET_NAME, OBJECT_ATTRIBUTE_NAME,
RELATION_SCHEME_NAME, RELATIONAL_ATTRIBUTE_NAME)
values("PERSON", "SSN", "PERSON", "SSN")
go
insert ATTRIBUTE_MAPPING(OBJECT_SET_NAME, OBJECT_ATTRIBUTE_NAME,
RELATION_SCHEME_NAME, RELATIONAL_ATTRIBUTE_NAME)
values("PERSON", "NAME", "PERSON", "NAME")
go
insert ATTRIBUTE_MAPPING(OBJECT_SET_NAME, OBJECT_ATTRIBUTE_NAME,
RELATION_SCHEME_NAME, RELATIONAL_ATTRIBUTE_NAME)
values("PERSON", "SSN", "FACULTY", "SSN")
go
insert ATTRIBUTE_MAPPING(OBJECT_SET_NAME, OBJECT_ATTRIBUTE_NAME,
RELATION_SCHEME_NAME, RELATIONAL_ATTRIBUTE_NAME)
values("FACULTY", "RANK", "FACULTY", "RANK")
go
insert ATTRIBUTE_MAPPING(OBJECT_SET_NAME, OBJECT_ATTRIBUTE_NAME,
RELATION_SCHEME_NAME, RELATIONAL_ATTRIBUTE_NAME)
values("DEPARTMENT", "NAME", "DEPARTMENT", "NAME")
go
insert ATTRIBUTE_MAPPING(OBJECT_SET_NAME, OBJECT_ATTRIBUTE_NAME,
RELATION_SCHEME_NAME, RELATIONAL_ATTRIBUTE_NAME)
values("PERSON", "SSN", "COURSE", "FACULTY_SSN")
go
insert ATTRIBUTE_MAPPING(OBJECT_SET_NAME, OBJECT_ATTRIBUTE_NAME,
RELATION_SCHEME_NAME, RELATIONAL_ATTRIBUTE_NAME)
values("DEPARTMENT", "NAME", "COURSE", "DEPARTMENT_NAME")
go
insert ATTRIBUTE_MAPPING(OBJECT_SET_NAME, OBJECT_ATTRIBUTE_NAME,
RELATION_SCHEME_NAME, RELATIONAL_ATTRIBUTE_NAME)
values("COURSE", "NUMBER", "COURSE", "NUMBER")
go
insert RELATION_KEYS(RELATION_SCHEME_NAME, Number, Type)
values("PERSON", 0, "PRIMARY")
insert RELATION_KEYS(RELATION_SCHEME_NAME, Number, Type)
values("FACULTY", 0, "PRIMARY")
insert RELATION_KEYS(RELATION_SCHEME_NAME, Number, Type)
values("FACULTY", 1, "FOREIGN")
go
insert RELATION_KEYS(RELATION_SCHEME_NAME, Number, Type)
values("DEPARTMENT", 0, "PRIMARY")
insert RELATION_KEYS(RELATION_SCHEME_NAME, Number, Type)
values("COURSE", 0, "PRIMARY")
insert RELATION_KEYS(RELATION_SCHEME_NAME, Number, Type)
values("COURSE", 1, "FOREIGN")
go
insert RELATION_KEYS(RELATION_SCHEME_NAME, Number, Type)
values("COURSE", 2, "FOREIGN")
go
insert RELATION_KEYS(RELATION_SCHEMA_NAME, Number, Type)
values("TEACH", 0, "PRIMARY")
insert RELATION_KEYS(RELATION_SCHEMA_NAME, Number, Type)
values("OFFER", 0, "PRIMARY")
insert KEY_ATTRIBUTE(RELATION_SCHEMA_NAME, RELATIONAL_ATTRIBUTE_NAME,
IN_RELATION_SCHEMA_NAME, IN_RELATION_KEYS_Number)
values("PERSON", "SSN", "PERSON", 0)
go
insert KEY_ATTRIBUTE(RELATION_SCHEMA_NAME, RELATIONAL_ATTRIBUTE_NAME,
IN_RELATION_SCHEMA_NAME, IN_RELATION_KEYS_Number)
values("FACULTY", "SSN", "FACULTY", 0)
go
insert KEY_ATTRIBUTE(RELATION_SCHEMA_NAME, RELATIONAL_ATTRIBUTE_NAME,
IN_RELATION_SCHEMA_NAME, IN_RELATION_KEYS_Number)
values("DEPARTMENT", "NAME", "DEPARTMENT", 0)
go
insert KEY_ATTRIBUTE(RELATION_SCHEMA_NAME, RELATIONAL_ATTRIBUTE_NAME,
IN_RELATION_SCHEMA_NAME, IN_RELATION_KEYS_Number)
values("COURSE", "NUMBER", "COURSE", 0)
go
insert KEY_ATTRIBUTE(RELATION_SCHEMA_NAME, RELATIONAL_ATTRIBUTE_NAME,
IN_RELATION_SCHEMA_NAME, IN_RELATION_KEYS_Number)
values("COURSE", "FACULTY_SSN", "COURSE", 1)
go
insert KEY_ATTRIBUTE(RELATION_SCHEMA_NAME, RELATIONAL_ATTRIBUTE_NAME,
IN_RELATION_SCHEMA_NAME, IN_RELATION_KEYS_Number)
values("TEACH", "COURSE_NUMBER", "TEACH", 0)
go
insert KEY_ATTRIBUTE(RELATION_SCHEMA_NAME, RELATIONAL_ATTRIBUTE_NAME,
IN_RELATION_SCHEMA_NAME, IN_RELATION_KEYS_Number)
values("OFFER", "COURSE_NUMBER", "OFFER", 0)
go
insert REFERENTIAL_INTEGRITY(TO_RELATION_SCHEMA_NAME, FROM_RELATION_SCHEMA_NAME,
FROM_RELATION_KEYS_Number, Insert_Rule, Delete_Rule, Update_Rule)
values("PERSON", "FACULTY", 1, "RESTRICTED", "RESTRICTED", null)
go
insert REFERENTIAL_INTEGRITY(TO_RELATION_SCHEMA_NAME, FROM_RELATION_SCHEMA_NAME,
FROM_RELATION_KEYS_Number, Insert_Rule, Delete_Rule, Update_Rule)
values("FACULTY", "COURSE", 1, "RESTRICTED", "RESTRICTED", null)
go
insert REFERENTIAL_INTEGRITY(TO_RELATION_SCHEMA_NAME, FROM_RELATION_SCHEMA_NAME,
FROM_RELATION_KEYS_Number, Insert_Rule, Delete_Rule, Update_Rule)
values("DEPARTMENT", "COURSE", 2, "RESTRICTED", "RESTRICTED", null)
go
insert CONNECTION_MAPPING(OBJECT_CONNECTION_Role, FROM_OBJECT_SET_NAME, TO_OBJECT_SET_NAME, TO_RELATION_SCHEME_NAME, FROM_RELATION_SCHEME_NAME, FROM_RELATION_KEYS_Number)
values("!NONE!", "FACULTY", "PERSON", "PERSON", "FACULTY", 1)

go
insert CONNECTION_MAPPING(OBJECT_CONNECTION_Role, FROM_OBJECT_SET_NAME, TO_OBJECT_SET_NAME, TO_RELATION_SCHEME_NAME, FROM_RELATION_SCHEME_NAME, FROM_RELATION_KEYS_Number)
values("!NONE!", "COURSE", "FACULTY", "FACULTY", "COURSE", 1)

go
insert CONNECTION_MAPPING(OBJECT_CONNECTION_Role, FROM_OBJECT_SET_NAME, TO_OBJECT_SET_NAME, TO_RELATION_SCHEME_NAME, FROM_RELATION_SCHEME_NAME, FROM_RELATION_KEYS_Number)
values("!NONE!", "COURSE", "DEPARTMENT", "DEPARTMENT", "COURSE", 2)

go
insert DOMAIN(NAME)
values("int")

go
insert DOMAIN(NAME)
values("char(50)"")

go
insert DOMAIN(NAME)
values("char(25)"")

go
insert DOMAIN(NAME)
values("char(30)"")

go
insert WITH_A(RELATION_SCHEME_NAME, RELATIONAL_ATTRIBUTE_NAME, DOMAIN_NAME)
values("PERSON", "SSN", "int")

go
insert WITH_A(RELATION_SCHEME_NAME, RELATIONAL_ATTRIBUTE_NAME, DOMAIN_NAME)
values("PERSON", "NAME", "char(50)"")

go
insert WITH_A(RELATION_SCHEME_NAME, RELATIONAL_ATTRIBUTE_NAME, DOMAIN_NAME)
values("FACULTY", "SSN", "int")

go
insert WITH_A(RELATION_SCHEME_NAME, RELATIONAL_ATTRIBUTE_NAME, DOMAIN_NAME)
values("FACULTY", "RANK", "char(25)"")

go
insert WITH_A(RELATION_SCHEME_NAME, RELATIONAL_ATTRIBUTE_NAME, DOMAIN_NAME)
values("DEPARTMENT", "NAME", "char(30)"")

go
insert WITH_A(RELATION_SCHEME_NAME, RELATIONAL_ATTRIBUTE_NAME, DOMAIN_NAME)
values("COURSE", "FACULTY_SSN", "int")

go
insert WITH_A(RELATION_SCHEME_NAME, RELATIONAL_ATTRIBUTE_NAME, DOMAIN_NAME)
values("COURSE", "DEPARTMENT_NAME", "char(30)"")

go
insert WITH_A(RELATION_SCHEME_NAME, RELATIONAL_ATTRIBUTE_NAME, DOMAIN_NAME)
values("COURSE", "NUMBER", "int")
go
insert GLOBAL_SUBJ_TERM(NAME)
values("Course_Teaching")

go
insert GLOBAL_SUBJ_TERM(NAME)
values("Course_Offering")

go
insert OBJ_SUBJ_TERM(NAME, Description)
values("Course_Teaching", "")

go
insert OBJ_SUBJ_TERM(NAME, Description)
values("Course_Offering", "")

go
insert OBJ_SUBJ_ASSOC(OBJ_SUBJ_TERM_NAME, OBJECT_SET_NAME)
values("Course_Teaching", "OFFER")

go
insert OBJ_SUBJ_ASSOC(OBJ_SUBJ_TERM_NAME, OBJECT_SET_NAME)
values("Course_Teaching", "TEACH")

go
insert OBJ_SUBJ_ASSOC(OBJ_SUBJ_TERM_NAME, OBJECT_SET_NAME)
values("Course_Teaching", "FACULTY")

go
insert OBJ_SUBJ_ASSOC(OBJ_SUBJ_TERM_NAME, OBJECT_SET_NAME)
values("Course_Offering", "OFFER")

go
insert OBJ_SUBJ_ASSOC(OBJ_SUBJ_TERM_NAME, OBJECT_SET_NAME)
values("Course_Offering", "COURSE")

go
insert OBJ_SUBJ_ASSOC(OBJ_SUBJ_TERM_NAME, OBJECT_SET_NAME)
values("Course_Offering", "DEPARTMENT")

quit
VI. THE PROGRAM STRUCTURE OF SDT

The program structure of SDT is shown in Figure 6.1, and consists of the following modules:

**main.c**  The main module reads the command line and accordingly invokes the functions in the following modules.

**parser.y**  The parser is written in YACC; it parses the input EER schema specification and builds the internal structure corresponding to the EER schema.

**scanner.l**  The scanner is written in LEX; it is used by the parser to read the input EER schema tokens.

**build.c**  This module contains the semantic functions used by the parser in order to build the internal structure representing the EER schema.

**wfcheck.c**  This module consists of functions used to check whether an EER schema is well-formed. This module also assigns a level number to each object representing the partial order of the object in the EER schema.

**mapping.c**  This module consists of the functions that comprise the mapping of an EER schema into an abstract relational schema in Boyce-Codd Normal Form.

**assign.c**  This module performs the assignment of local names to relational attributes. (Local names are unique only within a given relation-scheme.)

**convert.c**  This module reads the internal form of an abstract relational schema and prints it out in a readable form.

**merge.c**  Merging can be either skipped (no merging option) or performed.

**tosybase.c**  This module translates the abstract relational schema into its equivalent SYBASE schema, consisting of table, key, and index definitions, and trigger specifications.

**toisql.c**  This module translates the abstract relational schema into its equivalent INGRES 6.3/SQL schema, consisting of table and index definitions, and rule specifications.

**toinformix.c**  This module translates the abstract relational schema into an INFORMIX 4.0/SQL schema consisting of table and index definitions.

**veref_s.c**  This module generates SYBASE procedures for verifying the integrity of a SYBASE database with regard to the referential integrity constraints of the abstract relational schema.
Figure 6.1 The Structure of SDT.
**Program Structure**

- **veref_i.c**  This module generates INGRES procedures for verifying the integrity of an INGRES database with regard to the referential integrity constraints of the abstract relational schema.

- **metadb_s.c**  This module generates the metadata describing the EER and abstract relational schemas, and their mappings; these metadata can be loaded into a predefined SYBASE metadatabase.

- **metadb_i.c**  This module generates the metadata describing the EER and abstract relational schemas, and their mappings; these metadata can be loaded into a predefined INGRES metadatabase.

- **metadb_x.c**  This module generates the metadata describing the EER and abstract relational schemas, and their mappings; these metadata can be loaded into a predefined INFORMIX metadatabase.

- **common.h**  This file contains the definitions of constants, data types; and declarations of external variables and functions.

- **Makefile**  This file is the input to the *make* UNIX program; the execution of *make* generates the executable binary code. As a byproduct, this file also specifies how the various modules are related to each other.
REFERENCES


APPENDIX A. THE EXTENDED ENTITY–RELATIONSHIP MODEL

The concepts of the Entity–Relationship model have been defined originally in [6] and have been repeatedly reviewed since then. The Extended Entity–Relationship model is surveyed in [7]. We follow, in general, the definitions of [6] and [7], with slight modifications. Unlike [6] and [7], however, we represent Entity-Relationship structures by directed, rather than undirected, diagrams.

A.1 Fundamental Concepts

A.1.1 Object–Sets

The first stage of Entity-Relationship (ER) modeling consists of determining the principal objects about which information is collected, called entity-sets. Entity-sets are qualified by attributes, that represent their descriptive properties. For instance, PERSON could be an entity-set with attributes SOCIAL-SECURITY-NUMBER, NAME, JOB-TITLE, and SALARY. Associations of entity-sets are represented by relationship-sets. For instance WORK could be a relationship-set associating entity-sets PERSON and PROJECT. A relationship-set may have attributes, just like an entity-set, such as the PERCENTAGE-OF-TIME a person WORKS on each project. Individual instances of entity-sets and relationship-sets are called entities and relationships, respectively. In the following we shall refer commonly to entities and relationships as objects, and to entity-sets and relationship-sets as object-sets.

A.1.2 Value–Sets

Attributes take their values from underlying domains called value-sets. Examples of value-sets could be CHARACTER, INTEGER. Value-sets can be associated with a format describing the structure of their elements (e.g. six-digit character). Attributes provide an interpretation of a given value-set in the context of some object-set. For instance, attribute NAME gives the interpretation of value-set CHARACTER in the context of entity-set PERSON. The independent identity of attribute values is of no interest in the modeled environment, but only when coupled with some object. For instance a value of attribute SOCIAL-SECURITY-NUMBER is of interest only as characterizing an instance of entity-set PERSON. Value-sets are the basis of correlating attributes: attributes associated with the same value-set, are said to be compatible, that is, can be compared. When value-sets are uninterpreted, that is, devoid of any semantic meaning (e.g. sets of integers or characters), the attribute compatibility has no real significance. For instance, although two attributes, such as AGE and HEIGHT, could be based on a same value-set (e.g. numbers) their comparison could be meaningless. Value-sets can be interpreted by associating them with units. Then two attributes are said to be compatible only if the units of their underlying value-sets are the same or can be converted to a common unit. For instance the value-set underlying attributes AGE and HEIGHT could be associated with years and kilograms as units, respectively. Interpreted value-sets allow
the specification of two kinds of constraints: (i) *value constraints* restrict the values that an attribute can take from a value-set (e.g., the value-set of attribute AGE can be specified as consisting of integers between 13 and 65); and (ii) *operational constraints* restrict the operations allowed on the attribute values (e.g., AGE values could be added and subtracted, while NAMES values could be compared but not added). Generally attributes can be associated not only with single value-sets, but also with the cartesian product of several value-sets.

### A.1.3 Entity–Relationship Diagram

ER structures are expressible in a diagrammatic form called **ER Diagram** (ERD). Entity-sets, relationship-sets, and attributes, are represented graphically by rectangles, diamonds, and ellipses, respectively. Every vertex is labeled by the name of the object-set or attribute; entity, and relationship vertices are uniquely identified by their labels globally, while attribute vertices are uniquely identified by their labels only locally, with respect to their object-set (that is, within the set of attribute vertices connected to some object-set vertex). Edges in an ER diagram represent the interaction of the various object-sets and attributes. The ER diagram is a *directed graph*, that is, it has directed edges. In figure A.1 we present an example of an ER diagram consisting of the following main components: PERSON, PROJECT, DIVISION and DEPARTMENT are entity-sets, relationship-set EMPLOYED represents the employment of persons by departments, relationship-set ASSIGNMENT represents the assignment of projects to departments, and KINSHIP represents the kinship relation between persons.

### A.1.4 Entity–Identifier

A subset of the attributes associated with an entity-set is specified as the **entity–identifier**. Entity-identifiers are used to distinguish among the instances of an entity-set. For instance SOCIAL–SECURITY–NUMBER could be an identifier for entity-set PERSON, as shown in the ER diagram of figure A.1 where attributes belonging to identifiers are underlined. However, entity-identifiers are not always enough to *uniquely* distinguish among the instances of an entity-set. For example, there may be a Service department in both the Appliance and Automotive divisions of some company. In that case, the entity-identifier NAME of entity-set DEPARTMENT is not enough to uniquely distinguish between the various instances of departments with the same name in different divisions. Such entity-sets are called *weak*, and said to depend for identification (*ID–dependent*) on other entity-sets. In ER diagrams, vertices that represent weak entity-sets are connected by directed edges, labeled *ID*, to the vertices representing the entity-sets on which the weak entity-sets depend. For instance, in the former example DEPARTMENT could be made ID-dependent on entity-set DIVISION, as shown in the ER diagram of figure A.1. We assume that there is a *single* identifier specified for every entity-set, although other alternate identifiers can be also
specified.

A.1.5 Existence Dependency

ER structures imply certain **existence dependencies** among interacting objects. An object-set \( O_i \) is said to depend existentially on an object-set \( O_j \) if any object of \( O_i \) exists *only if* a related object of \( O_j \) also exists. Accordingly, relationships depend on the existence of the associated entities. For example, an ASSIGNMENT relationship can be specified only if the corresponding involved DEPARTMENT and PROJECT entities also exist. Similarly, weak entities depend on the existence of the entities needed for their identification. For example, a DEPARTMENT entity can be specified only if the corresponding DIVISION entity needed for its identification, also exists. In ER diagrams edges represent not only the interaction of the various ER objects, but also their mutual existence dependencies. Thus, there will be directed edges (i) from relationship-sets to the entity-sets they associate; and (ii) from weak entity-sets to the entity-sets on which they depend for identification.

A.1.6 Association and Involvement Cardinality

**Association-cardinality** and **involvement-cardinality** are restrictions placed on an entity-set with respect to a relationship-set. Association and involvement-cardinalities can be either *one* or *many*. For example, in the relationship-set EMPLOYED associating the DEPARTMENT and PERSON entity-sets, the DEPARTMENT entity-set would have an association-cardinality of *many* if each person is allowed to be employed in several departments, and of *one* if we wish to express the restriction of each person being employed in one department only. Conversely, the same restrictions are expressed by involvement-cardinalities of *many*, respectively *one*, of entity-set PERSON. Formally, if \( R_k \) is a relationship-set that involves entity-set \( E_i \), then (i) an association-cardinality of *one* for \( E_i \) in \( R_k \) means that, given any element of the cross-product of all the entity-sets involved in \( R_k \) except \( E_i \), there is *at most one* instance of

![Figure A.1 An Entity-Relationship Diagram Example.](image-url)
$E_i$ that can be associated by $R_k$ with that element; and (ii) an involvement-cardinality of one for $E_i$ in $R_k$ means that an entity of $E_i$ can be involved in at most one relationship of $R_k$. This definition applies to any relationship-set, irrespective of the number of entity-sets it associates.

In ER diagrams association and involvement-cardinalities are represented by labels. Thus, if entity-set $E_i$ has an association (resp. involvement) cardinality of one with respect to $R_k$, then the edge connecting the vertices representing $E_i$ and $R_k$ is associated with label 1 (resp. Inv1); and if entity-set $E_i$ has an association (resp. involvement) cardinality of many with respect to $R_k$, then the edge connecting the vertices representing $E_i$ and $R_k$ is associated with label $M$ (resp. InvM). Edges that connect entity vertices with relationship vertices, and that are not associated with such a label, are assumed to correspond to cardinalities of many. In the ER structure represented in figure A.1, for example, the involvement-cardinalities of relationship-set EMPLOYED represent the restriction of a person being employed by at most one department, and the involvement-cardinalities of relationship-set ASSIGNMENT represent the restriction of a project being assigned to at most one department.

A.1.7 Mandatory Involvement

The involvement of objects in relationships is, by default, optional. For example, the entities of entity-set PROJECT may or may not be involved in relationships of relationship-set ASSIGNMENT, which means that there could be projects that are not assigned to any department (e.g. because the department is not yet known). Conversely, the involvement of an object-set in a relationship-set can be specified as mandatory, which means that an object of that object-set must be involved, at any time, in at least one relationship of the respective relationship-set. Mandatory involvement of entity-sets in relationship-sets is represented graphically by double-line edges instead of the regular edges representing the non-mandatory (optional) involvements. For example, the mandatory involvement of entity-set PROJECT in relationship-set ASSIGNMENT is represented as shown in the ER diagram of figure A.1, and means that each project must be assigned, at any time, to at least one department.

A.1.8 Role

An entity-set involved in a relationship-set is said to have a role in that relationship-set. Roles are essential in distinguishing the multiple involvements of an entity-set in a relationship-set (represented in the corresponding ER diagram by parallel edges from the relationship-set vertex to the entity-set vertex). Roles are represented in ER diagrams by labels on the edges connecting the corresponding object-sets. For example, the two involvements of entity PERSON in relationship-set KINSHIP are characterized by distinct two roles, PARENT and CHILD, respectively, which are represented as shown in figure A.1.
A.2 Extended Concepts

The concepts of entity-set, relationship-set, attribute, and value-set are fundamental in the ER model. Two abstraction capabilities that were not included in the original ER model and have been subsequently added are generalization and aggregation. The ER model extended with generalization and aggregation is called the Extended ER (EER) model.

A.2.1 Generalization

Generalization emphasizes the similarities of entities, while abstracting away their differences. Thus, generalization views a set of entity-sets (e.g. employees, students, scientists, secretaries) as a single generic entity-set (e.g. persons). The attributes which are common to the entity-sets that are generalized (such as name and age) are then represented only once, associated with the generic entity-set. Similarly, relationship-sets that are common to the entity-sets that are generalized are associated with the generic entity-set. The entity-sets that are generalized can have additional attributes of their own (e.g. scientists can have degrees) and can be involved in relationship-sets in which the generic entity-set is not involved (e.g. scientists may be related to projects, while secretaries are not). The inverse of generalization is called specialization. A specialization entity-set inherits all the attributes of any of its generic entity-sets, including the entity-identifier.

A.2.2 Types of Generalization

Generalization can abstract either homogeneous or heterogeneous entity-sets. In the first case generalization is called homogeneous generalization, and in the second case generalization is called heterogeneous kind. For homogeneous generalization, the type of the generic entity-set unifies and replaces the types of the specialization entity-sets, while for heterogeneous generalization the type of the generic entity-set is a new virtual type and the types of the specialization entity-sets are preserved. While entity of any homogeneous-specialization entity-set is allowed to migrate to any other homogeneous-specialization of the same generic entity-set (that is, is allowed to change roles), entities of heterogeneous-specialization entity-sets are not allowed to migrate to any other entity-set. For instance, entity-sets STUDENT and EMPLOYEE can be homogeneous-generalized by generic entity-set PERSON; then a STUDENT entity is allowed to migrate to entity-set EMPLOYEE (i.e. a person can cease to be a student and become an employee, or be both a student and an employee). In contrast, entity-sets GOV.OFFICE and COMPANY can be heterogeneous-generalized by generic entity-set SPONSOR; then a COMPANY entity is not allowed to migrate to entity-set GOV.OFFICE (i.e. a company cannot 'become' a government office). Typically (but not necessarily), heterogeneous-generic entity-sets are required to be covered by their heterogeneous-specializations.
A.2.3 Extended Entity–Relationship Diagram

We must extend the definition of the ER diagram in order to represent the new generalization construct; the extended ER diagram is called EER diagram. The vertices representing specializations are connected by directed edges labeled ISA to the vertices representing the corresponding generic entity-sets; for heterogeneous-generalizations the edges are double-shafted and the label is ISA*.

The EER diagram of figure A.2 extends the ER diagram of figure A.1, with two generalization hierarchies, namely the PERSON homogeneous-generalization of entity-set EMPLOYEE and and the SPONSOR heterogeneous-generalization of entity-sets COMPANY and GOV. OFFICE. The second generalization allows the association of entity-set PROJECT with SPONSOR by relationship-set SPONSORS, which represents the sponsoring of projects by government offices and private companies. Without the generalization capability PROJECT would be associated by two different relationship-sets with the entity-sets GOV. OFFICE and COMPANY, respectively, although these relationship-sets express the same kind of association.

A.2.4 Role Revisited

Homogeneous-generalization implies the specification of new roles for the homogeneous-generic entity-sets. Thus, if entity-set $E_i$ is a homogeneous-specialization of entity-set $E_j$, then $E_i$ and $E_j$ assume two distinct roles in their involvements with other entity-sets or relationship-sets. For example, the involvement of entity-set PERSON in relationship-set EMPLOYED in figure A.1, is replaced in figure A.2 by the involvement of entity-set EMPLOYEE in EMPLOYED, so that only a PERSON in the role of EMPLOYEE is associated by relationship-set EMPLOYED.

A.2.5 Aggregation

Aggregation is intended as a construct that can be applied over previously aggregated objects as many times as one wishes. For example, suppose that a relationship-set ASSIGNMENT associates entity-sets PROJECT and DEPARTMENT, as shown in figure A.2. We wish to relate PERSON (EMPLOYEE) and ASSIGNMENT. We could define a ternary relationship-set WORKS between entity-sets PROJECT, DEPARTMENT, and PERSON, but then relationships of this ternary relationship-set could associate PROJECT and DEPARTMENT entities that are not associated by any ASSIGNMENT relationship, contrary to our intention. The obvious and natural solution is to specify the relationship-set WORKS between relationship-set ASSIGNMENT and entity-set EMPLOYEE, as shown in figure A.2. Note that no extension is needed for the EER diagram in order to accommodate this new aggregation construct because of the use of directed edges.
Figure A.2 An Extended Entity-Relationship Diagram Example.