Chapter 11

An Object-Oriented Active Data Dictionary to Support Database Evolution

Hans Weigand, Berrie Kremers

11.1 Introduction

The Multimedia database of SPRITE is the central repository in which all information concerning documents is stored, including the contents and logical structure. When SPRITE is used in some application environment, it must be possible to integrate new applications or customize the available ones. This is to be achieved by the extensibility function. Extensibility enables a database system to be modelled as an evolutionary object. The basic principles of achieving extensibility are modularity and reusability. It is generally recognized that the object- oriented approach can greatly help in achieving these extensibility goals (see chapter 11). In the same chapter, it is argued that implementation efficiency can be improved by adopting a hybrid organization, that is, a relation-based scheme is used on disk to speed up associative access and an object-based approach is used in main memory to facilitate manipulations of single objects. In the SPRITE database this is realized by means of an object- oriented layer around a commercially available relational system (Sybase - see chapter 7 for more details).

To support reusability of specifications of code, Sprite incorporates an Active *Data Dictionary*. The Data Dictionary (DD) is a database that stores the Conceptual Schema of the MMD Document Database. The Conceptual Schema is specified in an object-oriented style. Basically, it consists of object class definitions, where each class definition specifies the attributes and methods of that class. The DD allows the user to browse through this Schema and reuse the specifications by importing them into another Schema or by extending the Schema with new classes or subclasses. Such an extension can be represented in the DD as a new version of the Schema, since the DD supports versions of design objects. Next, the user/designer can recompile the Schema and generate (Sybase) database tables, constraints, triggers and procedures. It is possible to override the output if there is, for example, a specific efficiency

requirement. The actual output (including modifications) is stored in the DD as well. In this way, each recompilation knows exactly what is already in the database so that it generates only the new parts. At the moment, we only allow incremental changes, no destructive modifications. Hence we avoid many problems associated with schema updates (see e.g. Casanova et al, 1991; Banerjee et al, 1987) in order to concentrate on the manipulative power of the Data Dictionary and the generation of directly usable Sybase code.

This chapter is divided as follows. Section 2 gives a global overview of the Data Dictionary concepts. Next, section 3 describes the graphical interface of the DD, while compilation of database structures is presented by example in section 4. Finally, section 5 discuses some directions for future research.

11.2 Data Dictionary concepts

The Data Dictionary (DD) is a database that stores the Conceptual Schema of the MMD Document Database. The Conceptual Schema is specified in an object-oriented style. Basically, it consists of object class definitions, where each class definition specifies the attributes and methods of that class. Attributes can be single-valued or set-valued, and can be data objects (from an extensible set of primitive data types such as string and integer) or references to other objects. The former types are called LOTs (Lexical Object Types), as in the NIAM method, the latter NOLOTS, or just object classes. A special class of attributes is formed by the so-called aggregations. The difference between attributes and aggregations is that aggregations are considered as part of the object for the purpose of locking, versioning, authorization and deletion, although this functionality is not completely supported yet in the Sprite system.

It is possible to add subclasses to a class and inherit attributes and methods. Subclasses belonging together logically are grouped into *generalizations*. This also allows the querying for a subtype. For example, a generalization of person could be "sex", with domain values "male" and "female". To retrieve all female persons, we add the selection "WHERE sex = female", to retrieve the sex subclass of a person we add "sex" as an attribute to the GET list just as if it were a normal attribute. We make a difference between dependent and independent generalizations, depending on the storage method employed. Dependent subclasses are stored together with the superclass; this is particularly useful when the subclasses denote different roles. Independent subclasses are stored separately. This is the usual case when subclasses denote natural kinds and diverge in the numbers and types of extra attributes.

Methods are defined in the Data Dictionary by means of the arity, the type of parameters, pre- and postconditions, triggers, and uninterpreted descriptions (see Weigand, 1991 for more details). It is possible that some method *specializes* a more

general method by extending any of the method components. At the moment, only primitive methods are generated.

To support extensibility, it is essential that the Data Dictionary can maintain different versions of the Conceptual Schema. The SPRITE Data Dictionary distinguishes two levels, the *framework* and the *schema level.* A framework is for example "sprite". It consists of a set of schemata, where each schema consists of a set of classes. In the framework of "sprite", we can have a document schema, a document space schema etc. Both frameworks and schemata are versioned. Moreover, it is possible to *import* classes from one schema into another. Such imported schemata cannot be updated.
The Data Dictionary has the following functions:

documentation
Storing the complete Conceptual Model (classes and methods). From the DD, reports can be generated in a readable format. It is also possible to query the Data Dictionary directly, using the standard query language (SQL) or some query module especially designed for this purpose.
The important advantage of the structured format is that it is more easy to maintain the model, that is, add new classes, change names, change attributes etc. By exploring the use of cross- references and constraints, it is easier to keep the documentation consistent under updates.

database generation
Database tables and atomic stored procedures can be generated automatically. This makes the Data Dictionary into an active Data Dictionary, that shortens the gap between design and implementation. The generator works in an incremental fashion, that is, merely extending the Conceptual Schema leaves the existing database structures unaffected.

Objective C code generation
Generate the Objective C classes with some simple generic procedures. This is related to the previous point. If the design is intended to be implemented by means of an Objective C layer around a relational database, as in SPRITE, the Objective C class definitions can be derived from the Data Dictionary definitions.

A function of the Data Dictionary not implemented fully yet is:

method code generation
The next step in implementation support, after the database generation, is the code generation for the methods. In the Data Dictionary, methods are either functions or events. Functions can be mapped easily to SQL views or queries by compiling the object-oriented SQL version we have adopted to Sybase SQL. Event methods are

more difficult. Under certain restrictions on the format of the pre- and postconditions, is it possible to compile method specifications in TRANSACT-SQL code. At present, only atomic stored procedures are generated for creating and deleting objects.

Some important features of the Data Dictionary are:
- *completeness* - the Conceptual Schema can be stored in the Data Dictionary completely
- *consistency* - consistency checks on the Conceptual Schema are supported
- *graphical interface* - the Data Dictionary can be consulted and updated by an easy to use graphical interface

11.3 Graphical user interface

For entering data into the data dictionary, a graphical user interface is created. This is done using the YILDIZ tool, an existing user interface generator system. A prototype can be made with it in a short time. The user interface consists of ten different windows, each for a separate concept in the data dictionary. We will show each window with a description. This will also make clear which concepts of the data dictionary are currently available.

When the user interface is started, the main window will appear on the screen (see fig. 11.1).

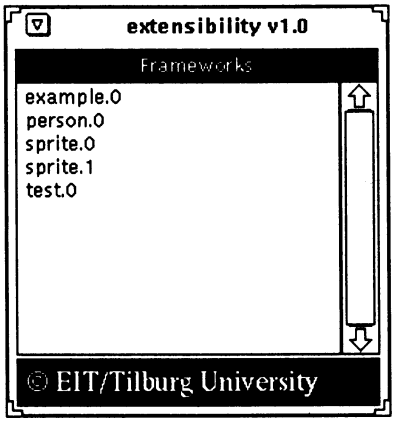

Fig. 11.1 Extensibility window

The window holds a menu containing all the frameworks in the data dictionary. Via a popup menu a new empty framework can be added, a new version of an existing

framework can be made, a framework can be deleted or a framework can be opened. Making a new version of a framework means that new versions are made of all the schemas contained in the framework.

If a framework is opened, a new window comes up (see fig. 11.2).

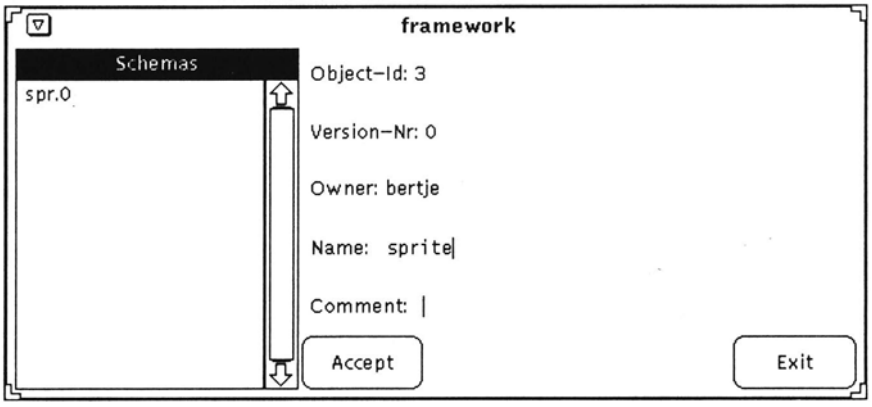

Fig. 11.2 Framework window

In the Schemas menu all the schemas contained in this framework are shown. The object-id, version-nr and owner of the framework are shown, they cannot be changed. The name of the schema and the comment on the schema are shown and editable. Changes made here are made permanent in the data dictionary when the Accept button is clicked. Clicking the Exit button will remove the window and all it's subwindows. Via a popup menu defined on the schemas menu, a new empty schema can be added, a schema can be removed or a schema can be opened. To make a new version of a schema, there are copy-paste entries in the popup menu. 'Copy' remembers the selected schema, 'paste-new-version' makes a new version of the remembered schema. This pasting can be done in another framework window. Sharing of schemas between frameworks is currently not supported yet.

If a schema is opened, the schema window appears (see fig. 3).

The fields on the right have the same function as the ones in the framework window. The Object Types menu contains all the object classes, LOTs and enumerations of this schema. LOTs are lexical object types, that is, data-valued types such as strings, whereas enumerations are sets of classes with a finite set of atomic instances, for example, "sex" with instances "male" and "female".

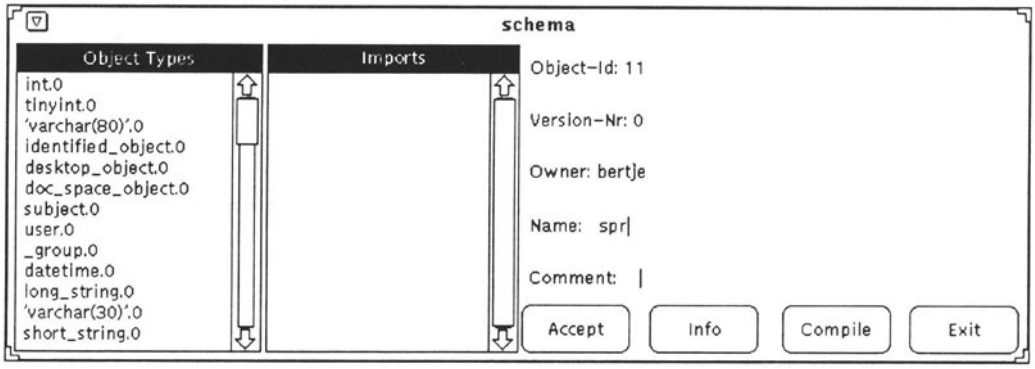

Fig. 11.3 Schema window

The Imports menu contains all the classes imported into this schema. Importing classes into a schema is done by using the copy paste mechanism (copy from another schema window, paste into this window). A popup menu is defined on the Object Types menu, via which new object types can be added (on creation one can choose the kind (LOT, enumeration or class), and the supertype (if any)), object types can be deleted, a new version of an object type can be pasted (after being copied from another schema window), and object types can be opened. On the right side there are two additional buttons, Info and Compile. Clicking the Info button will generate a report of the contents of the schema in readable form into the file 'info'. Clicking the Compile button will pop up a window, in which the names for the four output files can be chosen. One for the file into which the Sybase tables, keys and indexes will be generated, one into which Sybase triggers will be generated, one into which Sybase stored procedures will be generated, and one for '#define' statements generated from enumerations and generalizations, which can then be used in C code for example. Pressing 'Ok' starts the generation of code. Imported classes are not yet taken into account, neither for 'Info' nor for 'Compile'. When an object type is opened, one of three windows will popup, depending on the kind of object type.

The first one is the class window (see fig. 11.4).

The class window has four menus, one for the attributes, one for the keys, one for the generalizations and one for the tables. On each of the first three menus a popup menu is defined through which entries can be added, removed or opened. When an attribute is added, the domain of the attribute has to be chosen out of all the object types defined in the schema in which the class is contained. An attribute name may only occur once in a class. For adding a generalization a subclass of the class has to be specified (others can be added later in the generalization window, see fig. 11.8). For adding a key one attribute has to be specified (again others can be added in the key window, by copy/paste, see fig. 11.9). The tables menu is initially empty, it will get entries after a class has been compiled once. There will be an entry for each table that

was generated in the compilation. It is not possible to add or remove a table, it can only be opened.

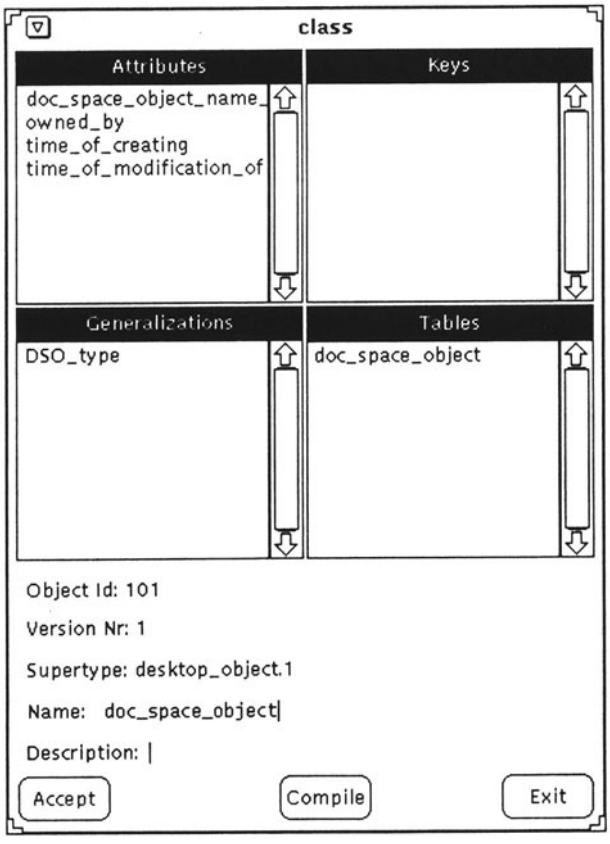

Fig. 11.4 Class window

Clicking the Compile button will popup a window, in which the names for the four output files can be chosen. One for the file into which the Sybase tables, keys and indexes will be generated, one into which Sybase triggers will be generated, one into which Sybase stored procedures will be generated, and one for '#define' statements generated from enumerations and generalizations, which can then be used in C code for example. Pressing 'Ok' starts the generation of code.

The second window reachable from the Schema window is the LOT window (see fig. 11.5):

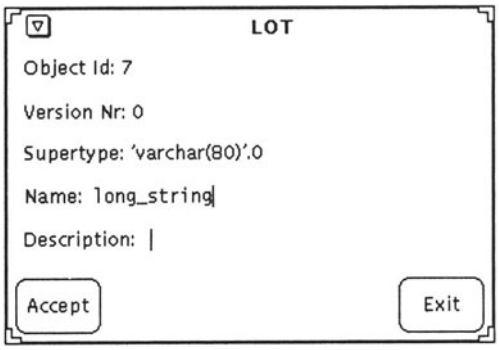

Fig. 11.5 LOT window

This is a very simple window, the only thing that can be done here is to change the name or the description of the LOT. The enumeration window looks very similar (see fig. 11.6).

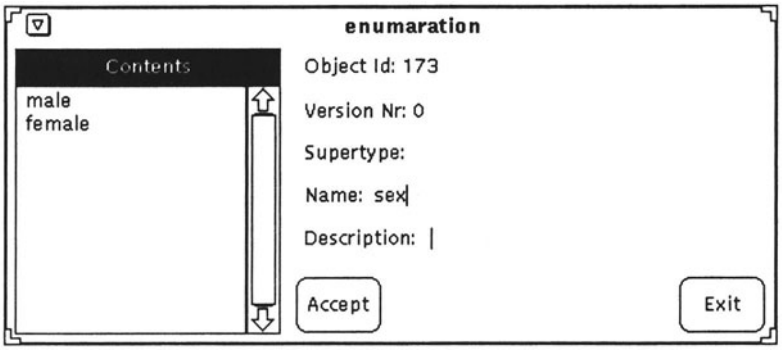

Fig. 11.6 Enumeration window

In addition this window has a menu of contents. Via a popup menu contents can be added or removed. When an attribute is opened from a class, the attribute window (see fig. 11.7) comes up.

The name and invert name of the attribute can be changed (the invert name is not used anywhere yet). When the Set button is clicked, a popup menu appears through which another domain for the attribute can be chosen. The attribute can be marked to be total (i.e., no nulls allowed) and/or set- valued. All changes are made permanent in the data dictionary only when the Accept button is clicked.

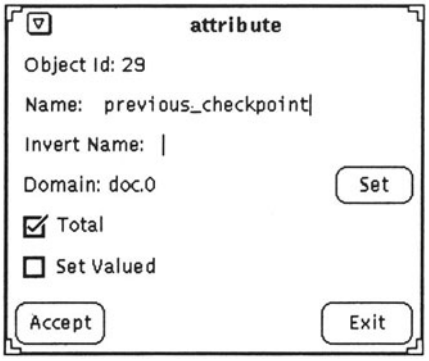

Fig. 11.7 Attribute window

When a generalization is opened, the generalization window comes up (see fig. 11.8).

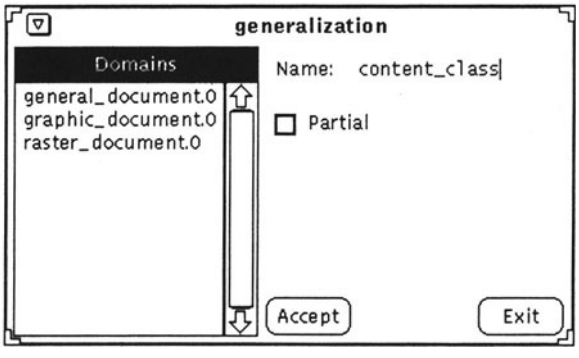

Fig. 11.8 Generalization window

The name of the generalization can be changed, and the generalization can be marked as being partial or not. The default is a that the subclasses makes up a total cover.

Domains can be added by copying an object type from the schema window (see fig. 11.3) that contains the class containing the generalization, and pasting it in the Domains menu. A domain of a generalization must be a subclass of the class that contains the generalization.

When a key is opened, the key window comes up (see fig. 11.9).

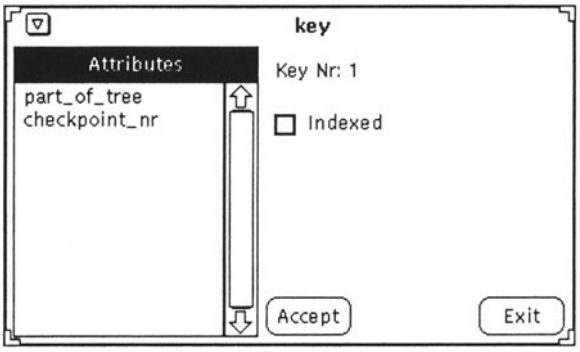

Fig. 11.9 Key window

Since there may be several (candidate) keys defined on a class, the keys are numbered by the system. The key nr is shown, but cannot be changed. Attributes can added to a key by pasting them after they have been copied form the class window (see fig. 11.6) containing the key. Such an attribute must be single-valued and total. The key can be marked to be indexed or not, but currently this has no effect on the mapping; it should result in a database (secondary) index on the attributes that improves access speed.

Finally, the last window of the user interface is the one for database tables (see fig. 11.10). It comes up when a table is opened.

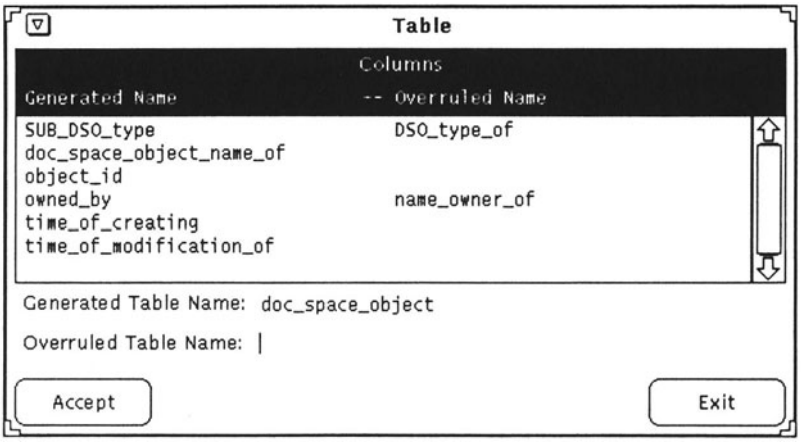

Fig. 11.10 Table window

This window shows a table as it was generated by the extensibility. The menu contains all the columns of the table. On the left hand side is the name generated by the extensibility, on the right hand side is the overruled name for the column (if set).

Via a popup menu a column name can be overruled, or an overruled name can be removed. By overruling, the designer can get exactly the names he wants in the database definition.

11.4 Compilation

When the Compile button in a schema window (see fig. 11.3) is pressed, the contents of that schema are mapped into Sybase table definitions, Sybase triggers and Sybase stored procedures. This section illustrates how this is done using a number of examples. All the parts that are in *italics* are generated.

LOTs and enumerations are mapped first. The mapping of LOTs is very simple. If a LOT has no superclass, nothing is done. In this way Sybase defined types (like int, tinyint or varchar(30)) can be defined in the system. If a LOT does have a superclass, a 'sp_addtype' is generated:

```
LOT longstring
ISA 'varchar(80)'
```

leads to

sp_addtype longstring, 'varchar(80)'
go

Only LOTs whose superclass is already mapped will be mapped, to avoid a reference to a non- existing type.

For enumerations, the supertype is not taken into account, although it can be specified. An enumeration is always mapped to type int. A rule is created for the type, to restrict the values to a subset of integers. The name of that rule is a concatenation of 'R_' and the name of the enumeration. When the mapping of a schema starts, enumeration definitions are read from the definition file that is specified before the compilation starts. All distinct contents of enumerations are mapped to distinct values, equal contents are mapped to equal values (if there are two enumerations containing the content 'xyz', this will map to the same value in the two enumerations). At the end of the mapping, new enumeration contents are added to the definition file.

```
ENUMERATION sex
CONTENTS {male, female}
```

leads to

```
sp_addtype sex, int
go

CREATE RULE R_sex
AS @sex in
(
        8       /* male */
,       9       /* female */)
go
```

and in the definition file to

```
#define male 8
#define female 9
```

Whenever a column in a table has a type that is an enumeration, a rule is bound to that column, to apply the rule.

```
CREATE TABLE person
(
      name          longstring    NOT NULL,
      sex           sex           NOT NULL
      )
```

gets a rule bound to the sex column

```
sp_bindrule R_sex, "person.sex"
go
```

After these two object types, the tables for the classes are generated. This includes the generation of foreign keys, primary keys and unique indexes. In the first step, where the objects of the generation classes are created, a separate object is made for each single table. For the actual generation, each time a table is chosen that has no unresolved references. This means that the table for its superclass is generated, and that the tables to which foreign keys of this table point, are also generated. If there are still tables that have to be generated, but all of them have unresolved references (this is possible if the references loop, for example a user must have a folder as homefolder, a folder must have a user as owner), the table that is referred to mostly is taken. The foreign keys that cannot be generated are kept, and will be generated as soon as possible (when both tables involved in the key are generated).

All the single-valued attributes of a class are mapped into one table. The name of the table is the class name. If the class has a superclass, the key attributes of the

superclass are mapped into that same table. If an attribute type is of kind LOT or enumeration, the table will get a column with the name of the attribute. The type of the column is the name of the LOT or enumeration that is the type of the attribute. Total is mapped to NOT NULL.

```
CLASS person
ATTRIBUTES
        name : longstring     total
        sex : sex             total
KEYS
        {name}
```

leads to

```
CREATE TABLE person
(       name    longstring      NOT NULL
,       sex     sex             NOT NULL
)
go
```

```
CLASS employee
ISA person
ATTRIBUTES
        company : longstring total
```

leads then to

```
CREATE TABLE employee
(       name    longstring  NOT NULL
,       company longstring  NOT NULL
)
go
```

If the type of the attribute is not a LOT but of kind class (we call this type class), there are two possibilities. If the type class has a key that is composed from only one attribute, then the name of the attribute will appear as a column in the table. If the key of the type class is composed of more attributes, then for each attribute part of the key, a column is added that is the concatenation of the attribute name and the type class key attribute name.

Each generalization is mapped into a single column of the table. The name of the column is a concatenation of 'SUB_' and the generalization name. The type of the

column is int, and a rule is created as for enumerations. A partial generalization is mapped to NULL, others to NOT NULL.

```
CLASS person
ATTRIBUTES
        name : longstring     total
        sex : sex             total
GENERALIZATIONS
        state_of_life {baby, child, adult, elderly}
                              partial
```

leads to

```
CREATE RULE R_state_of_life
AS @state_of_life in
(
          15        /* baby */
,         16        /* child */
,         17        /* adult */
,         18        /* elderly */
)
go

CREATE TABLE person
(         name                longstring  NOT NULL
,         sex                 sex         NOT NULL
,         SUB_state_of_life   int         NULL
)
go

sp_bindrule R_state_of_life, "person.SUB_state_of_life"
go
```

and in the definition file to

```
#define baby 15
#define child 16
#define adult 17
#define elderly 18
```

For every set-valued attribute, a separate table is created. The name of the table is the concatenation of the class name and the attribute name. All the columns are always NOT NULL.

For each table one or more keys are generated. There is always a primary key. If the class has a superclass, this key is the same as the one of the superclass. If the class has no superclass, the primary key is the key with the smallest number (this is the key that is created first for the class). All the other keys are not taken into account during the mapping. For the primary keys, corresponding indexes are generated.

Foreign keys are generated for three cases. First for subclass-superclass relations, a key from the subclass to the superclass on the key columns is created.

```
CLASS person
ATTRIBUTES
        name : longstring     total
        sex : sex             total
KEYS
        {name}

CLASS employee
ISA person
ATTRIBUTES
        company : longstring total
```

leads to

```
sp_foreignkey employee, person, name
go
```

The other cases where foreign keys are generated is for the relation between the 'main' table and set tables of a class and when the type of an attribute is of kind class (both for single- and set-valued attributes).

After the generation of the tables, keys and indexes, the mapping continues with the generation of Sybase triggers. Triggers are created for all foreign key relations, both on insert/update (the same trigger can be used for these two operations) and delete. The triggers check on each operation whether referential integrity is not violated, and if it is, they rollback the transaction. With the complete and correct set of triggers stored in a database, its integrity is maintained under all circumstances. The triggers generated by the extensibility make sure that there is never a reference to a non-existing object.

Each table has zero, one or two triggers. A class that has no superclass, and that has no attributes with a type that is of kind class, has no insert/update trigger. A class that has no subclasses, and that is not used as a type of some attribute, has no delete

trigger. Insert/update triggers start with a check if there is actually anything inserted or updated. After this there is a check for existence of the superclass object (if any), including a check on the generalization value. Then for each attribute that has a type of kind class a check for the existence of the referred object is added. If the attribute can have a NULL value, the check is extended to make it accept NULL values. The third code part generated by the extensibility consists of stored procedures for basic operations on objects. At this moment, insert and remove procedures are generated, one of each for each table. First the compiler looks at the procedures generated for a class that has no superclass. Two procedures are generated for the main table (containing all the single-valued attributes) and two for each set-table. The name of an insert procedure is CRF$ concatenated with the table name, the name of a remove procedure is FGT$ concatenated with the table name. The FGT$ procedures take as parameters the columns that build the key.

```
CLASS person
ATTRIBUTES
        name : longstring     total
        sex : sex             total
        spouse : person
        child :set of person  total
KEYS
        {name}
```

```
CREATE PROCEDURE CRF$person
        @m_name longstring,
        @m_sex sex,
        @i_spouse longstring = NULL
AS
        insert into person(name, sex, spouse)
        values
        (@m_name, @m_sex, @i_spouse)
go

CREATE PROCEDURE FGT$person
        @m_name longstring
AS
        delete from person where
                name = @m_name
go
```

If a class has a superclass the CRF$ and FGT$ procedures for the main table are extended to include inserting in or deleting from the superclass table as well. This is a

recursive process, if the superclass again has a superclass, its table is included as well, etc. etc. For set-tables nothing changes. The generalization columns for the superclasses are filled automatically.

11.5 Conclusion and directions for future work

The Active Data Dictionary of Sprite supports evolutionary design of the database and reusability. It uses an object-oriented specification language and a RDBMS as target system. The advantage of this approach is that it combines the efficiency of RDBMS storage structures and the modularity of Object-Orientation. As such it is also of potential benefit for users who want to migrate stepwise from a relational to an object-oriented environment.

Some directions for future work:

- *generation of method code*
  the DD gives the possibility to specify methods in a high-level language by means of pre- and postconditions and triggers. For many applications, this language is expressive enough. Generation of TRANSACT-SQL code (Stored Procedures) is not difficult, but still has to be done. In this way, we would circumvent one the serious drawbacks of Sybase Stored Procedures, namely, the lack of an adequate software environment, debugging tools etc.
- *integration with RIDL**
  the Data Dictionary can work in cooperation with the RIDL* tool (De Troyer, 1989). This CASE tool can generate database tables and constraints on the basis of NIAM pictures, a kind of Binary Relationship model. RIDL* has been applied successfully in the Sprite project. It is possible to design the database by means of NIAM and use RIDL* to generate the database schema. The Data Dictionary can then be applied afterwards for incremental extensions. Since the naming scheme of RIDL* is sometimes different from the one employed by the Data Dictionary compiler, the Dictionary gives the possibility to overrule the generated names. In the future, we would like to connect RIDL* more closely to DD, for example by making RIDL* generate DD specifications.

References

Banerjee J., Kim W. et al (1987) *Semantics and implementation of schema evolution in object- oriented databases.* ACM SIGMOD 1987, pp.311-322.

Casanova M.A. et al, (1991) *A Software Tool for Modular Database Design.* ACM TODS 16,2, pp.209-234.

De Troyer O. (1989) *RIDL*: A Tool for the computer-assisted engineering of large databases in the presence of integrity constraints.* ACM SIGMOD 1989, pp.418-429.

Weigand H. (1991) *An object-oriented approach in a multimedia database project.* In: Meersman, R.A. et al (eds), Object-oriented databases: Analysis, Design & Construction (DS- 4), North-Holland, Amsterdam, pp.393-413.