A Generator of SQL Schema Specifications

Slavica Aleksić[1], Ivan Luković[1], Pavle Mogin[2], Miro Govedarica[1]

[1] University of Novi Sad, Faculty of Technical Sciences,
21000 Novi Sad, Trg Dositeja Obradovića 6, Serbia
{slavica, ivan, miro}@uns.ns.ac.yu
[2] Victoria University of Wellington,
Wellington, P.O. Box 600, New Zealand
pmogin@mcs.vuw.ac.nz

Abstract. IIS*Case is an integrated CASE tool that supports the automation and intelligent support of complex and highly formalized design and programming tasks in the development of an information system. IIS*Case generates relational database schemas in 3rd normal form with all relevant data constraints. SQL Generator is an IIS*Case tool that generates the implementation specification of a database schema according to ANSI SQL:2003 standard. The generator may also produce a database schema specification for Microsoft SQL Server or Oracle DBMSs. The paper describes SQL Generator's traits, considers aspects of its application, and shows its use in the implementation of a complex database constraint using procedural mechanisms of a particular relational DBMS. SQL Generator is implemented in Java and Oracle JDeveloper environment.

1 Introduction

Integrated Information Systems*Case (IIS*Case) V.6.2 is a tool that provides the automation and an intelligent support for performing complex and highly formalized design and programming tasks in the development of an information system. It is designed to provide complete support for: (i) developing database (db) schemas that are complex with regard to the number of concepts used, and (ii) software applications of an information system.

The form type is one of the main IIS*Case concepts. It is semantically rich enough to enable expressing the elements of the static and dynamic structures in an application domain. By means of form types, a designer creates simultaneously a model of the structure and behavior of various business documents, and a conceptual db schema. Starting from the created conceptual schema, IIS*Case automatically generates a relational db schema in 3rd normal form (3NF) with all relevant data constraints. Detailed information about IIS*Case and its main concepts may be found in several authors' references, as well as in [5, 8, 13, 14].

IIS*Case is designed to provide a fast generation of db schemas and application prototypes. Although the concept of the form type is highly formalized, it is very close to the perception power of an average user. Therefore, IIS*Case may support an intensive and efficient communication among designers and users of an application domain throughout the software development process. Consequently, we believe that it is a tool suitable for the application in the agile software development. A case study illustrating main features of IIS*Case and the methodological approach to its usage is given in [5].

SQL Generator is a tool of IIS*Case that generates SQL specifications of relational db schemas. One of the main reasons for the development of such a tool was to make db designer's and developer's job easier, and particularly to free them from manual coding and testing of SQL scripts. The goal was to provide an efficient transformation of design specifications into error free SQL specifications.

There are a number of CASE tools that provide generation of SQL scripts. Some of them are described in [2, 12, 15]. One of the advantages of our SQL Generator is that it provides the implementation of some special cases of db constraints. For example, in contrast to Oracle Server Generator [12], and Sybase Power Designer PDM [15], IIS*Case SQL Generator provides the implementation of not only the default, but also the partial and the full referential integrity constraints, according to [4]. For all those types of referential integrity constraints, SQL Generator also allows selecting the following actions: No Action, Cascade, Set Default and Set Null, both for deleting and updating in the referenced table. Besides, SQL Generator provides the implementation of the inverse referential integrity constraints [8, 9], which are not rare in the real world and to the best of our knowledge, neither of the other CASE tools provides the same functionality. SQL Generator also produces a trigger that prohibits updates of a relation scheme primary key, if such a rule is specified in the design.

SQL Generator provides creating SQL scripts according to the syntax of: (i) ANSI SQL:2003 standard [4], (ii) DBMS Microsoft (MS) SQL Server 2000/2005 with Microsoft T-SQL [6, 7], and (iii) DBMS Oracle 9i/10g with Oracle PL/SQL [10, 11].

In this paper we present basic features of SQL Generator that are already implemented, and aspects of its application. We also present methods for implementation of a selected db constraint, using mechanisms provided by a relational DBMS. A complete description of SQL Generator may be found in [1].

2 Generating the SQL Specifications of a DB Schema

IIS*Case generates 3NF relational db schemas with all the relation scheme keys, null value constrains, unique constrains, referential and inverse referential integrity constraints. These schemas are stored in the IIS*Case repository. The specification of the IIS*Case repository is given in [13]. The input into SQL Generator is a schema stored in the repository.

Using SQL Generator, a user may produce SQL scripts for the creation of tables, views, indexes, sequences, procedures, functions and triggers, even without knowing SQL syntax and mechanisms for the implementation of constrains of a selected DBMS. SQL Generator may produce scripts for implementing a new db schema, or modify an already existing one in the following three ways: (i) by creating SQL scripts in files only for a later execution, (ii) by creating and immediately executing SQL scripts under a selected db server with an established connection, and (iii) by creating and immediately executing SQL scripts on a selected data source with an established connection via an ODBC driver. In all three cases, generated SQL scripts are stored in one or more files.

Figures 1-5 present screenshots of a form that is used to define values of SQL Generator input parameters. The field *DBMS* (Fig. 1) enables the selection of the type and version of a target db server. The radio button *DDL Files only* (Fig. 1) provides the creation of SQL scripts in files only. The scripts may be created in one, or more files (see the check box *One File Only* in Fig. 1). If a user selects the former option, separate files are created for tables, constraints, triggers and indexes. The main command file is also generated. It contains calls to all the other script files. The radio button *Database Source* (Fig. 1) enables the selection of either Oracle or MS SQL db server, establishing a connection, and immediate execution of SQL scripts. In this case, SQL Generator creates a script file, invokes the appropriate SQL tool, and passes necessary parameter values for the script execution. The radio button *ODBC Source* (Fig. 1) enables the creation and the immediate execution of SQL scripts in a selected ODBC data source. An appropriate ODBC driver for the target db server must be installed and configured. SQL Generator supports the user authentication when it works via an established connection. The field *DB Schema Name* (Fig. 1) enables defining a db name that is then included in an appropriate CREATE DATABASE command.

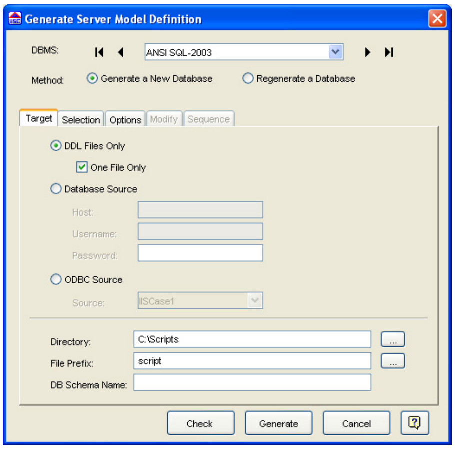

Figure 1. SQL Generator - Target panel

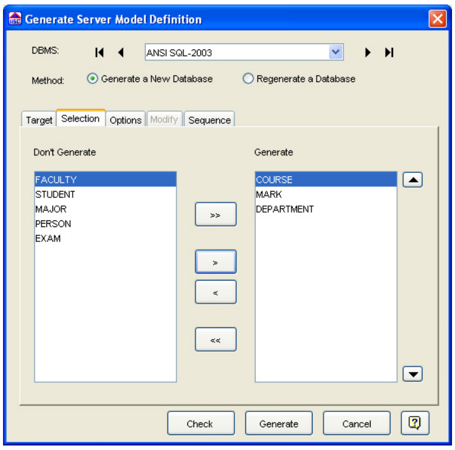

Figure 2. SQL Generator – Selection panel

By means of the *Selection* panel (Fig. 2), a user picks relation schemes. SQL Generator will produce the appropriate SQL commands for the selected relation schemes only, and place them in script files. The list of selected relation schemes should not be empty. Otherwise, a user will get a warning and the focus will be returned to the *Selection* panel automatically.

By means of the *Options* panel (Fig. 3), a user defines which types of db objects are to be generated. By checking the appropriate check-box items (Fig. 3), he or she may decide to generate: (i) indexes for primary, alternate and foreign keys, (ii) SQL CONSTRAINT clauses, (iii) triggers, and (iv) comments.

If the *Generate SQL CONSTRAINT Clauses* check-box is checked, SQL Generator produces the following CONSTRAINT clauses: PRIMARY KEY, UNIQUE, CHECK and FOREIGN KEY, for each key, unique, tuple, or foreign key constraint in db schema that may be implemented in a declarative way. If *Generate Triggers* check-box is checked, SQL Generator will produce all the db triggers, procedures and functions, necessary to implement db constraints that cannot be expressed in a declarative way. For inverse referential integrity constrains [8, 9], SQL Generator offers two ways of implementation: (i) by means of SQL views and the appropriate stored procedures, or (ii) by means of stored procedures only. If *Include Comments* checkbox is checked, SQL Generator will create comments in SQL code, comprising creation date and time, and the selected type and version of the DBMS.

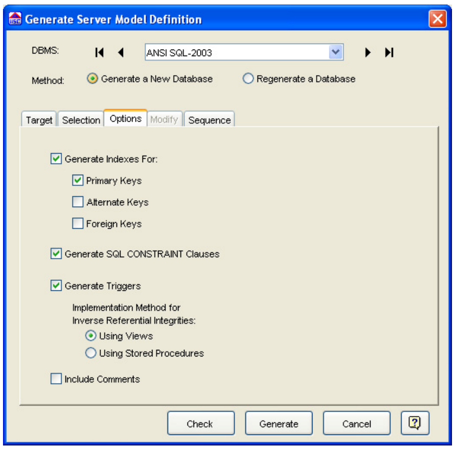

Figure 3. SQL Generator - Option panel

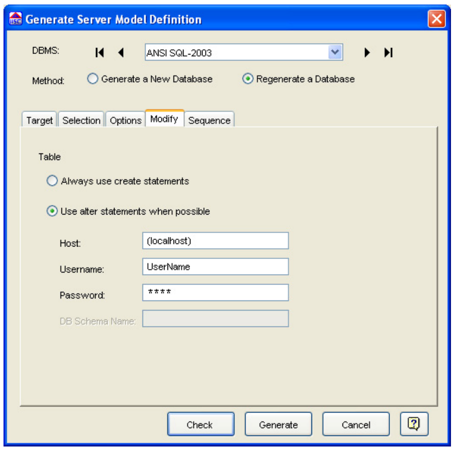

Figure 4. SQL Generator - Modify panel

A user can select one of the following script generating methods. If the radio button *Generate a New Database* is selected (Fig. 1-5), SQL Generator will produce scripts with CREATE statements, for the implementation of a new db schema under a DBMS. Otherwise, if *Regenerate a Database* is selected, it will regenerate, i.e. modify an already implemented db schema under a DBMS. In the later case, the *Modify* panel (Fig. 4) is used. SQL Generator uses the values for *Host*, *Username*, *Password* and *DB Schema Name* (Fig. 4) to establish a connection to the target database, compares the information from IIS*Case repository with the information obtained from the data dictionary of the target database, and generates scripts containing the appropriate CREATE, ALTER and DROP statements. If a user selects the radio button *Always use create statements*, each modification is accomplished by dropping old and then creating objects. Otherwise, if *Use alter statements when possible* is selected, SQL Generator will produce ALTER statements, whenever it is possible.

SQL Generator also supports the creation of sequence generators. A sequence generator specification is defined by the *Sequence* panel (Fig. 5). The selection of sequence generator properties, and the way of its implementation depends on the characteristics of the DBMS selected.

Not all possible combinations of the selected generator options are always valid. By pressing the *Check* button (Fig. 1-5), a user initiates a check of the selected options. If some inconsistencies arise, a user gets the appropriate warnings. Pressing the button *Generate* initiates the generation of SQL scripts and their saving in one or more files. The content of each generated file can be viewed, or modified through the form presented in Fig. 6. By pressing the button *Execute*, a user can also start the execution of a script file on a selected DBMS manually.

Generating SQL scripts may produce various kinds of warnings as a result of potentially incorrect designer's decisions. For example, the following will produce a warning: choosing a *Set Null* action for a constraint comprising a not null attribute, or giving names that will cause the name of a trigger longer than 30 characters in an Oracle DBMS. A separate panel *Messages* is used to view the warnings.

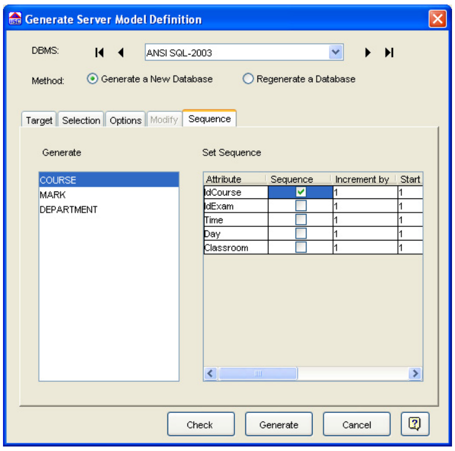

Figure 5. SQL Generator - Sequence panel

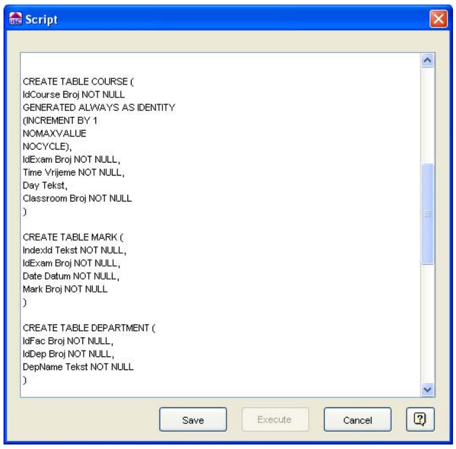

Figure 6. The form for reviewing script

3 Constraint Types Supported

SQL Generator implements constraints of the following types: domain constraints, key constraints, unique constraints, tuple constraints, native and extended referential integrity constraints (default, partial, full), referential integrity constraints inferred from nontrivial inclusion dependencies (default, partial, full), native inverse referential integrity constraints, and inverse referential integrity constraints inferred from nontrivial inclusion dependencies. [1, 3, 4, 8, 9]

According to [4], a designer qualifies each referential integrity constraint in IIS*Case as a default, partial or full and this affects the way of its validation. He or she also selects an action for preserving consistency in the case of an attempt to violate the constraint during inserts, updates, or deletes. The possible actions are: No Action, Cascade, Set Default and Set Null. Before implementing a constraint, SQL Generator analyzes designer's selections. If a selected combination is not applicable, SQL Generator produces a warning.

Constraints are implemented by the declarative DBMS mechanisms, whenever it is possible. However, the expressivity of declarative mechanisms of commercial DBMSs is usually limited in comparison to [4]. Therefore, SQL Generator implements a number of constraints through the procedural mechanisms.

4 An Example of the Procedural Implementation of a Constraint

Common algorithms for controlling a constraint validation are given in [1, 3, 8]. The process of the procedural implementation of a constraint can be unified. It consists of the following steps: (i) specifying a parameterized pattern of the algorithm for a specific DBMS, (ii) replacing the pattern parameters with real values, and (iii) generating an SQL script comprising necessary triggers, procedures and functions. [1]

In this Section, we present an example of a trigger for the control the deletion of a set of tuples from a relation $r(N_j)$, where N_j is a relation scheme participating in a native, partial referential integrity constraint $N_i[X] \subseteq N_j[Y]$. If a user selects ANSI SQL as a target DBMS, the trigger is not needed, since the partial referential integrity constraint is implemented declaratively, by means of the constraint clause FOREIGN KEY and its subclause MATCH PARTIAL [4]. Suppose a user selects MS SQL Server as a target DBMS. Since MS SQL Server currently does not support the MATCH clause, a trigger is needed. A parameterized generic pattern of such a trigger is shown in Fig. 7. A corresponding pattern for the Oracle Server is presented in [1].

```
CREATE TRIGGER TRG_<Nj>_<ConstraintName>_DEL ON <Nj> FOR DELETE
AS
 DECLARE   <DeclarationFor_Y>, <DeclarationFor_X>,  <DeclarationFor_PK_u>
 DECLARE Cursor_<Nj>  CURSOR FOR SELECT <AttributeSetFrom_Y> FROM Deleted
 OPEN Cursor_<Nj>
 FETCH NEXT FROM Cursor_<Nj>  INTO  <VariablesFor_Y>
 WHILE @@FETCH_STATUS=0
 BEGIN
    DECLARE Cursor_<Ni> CURSOR FOR
    SELECT<AttributeSetFrom_X>, <AttributeSetFrom_PK_u>  FROM <Ni>
    WHERE   <SelectionCriteria_ Cursor_Ni>
    OPEN Cursor_<Ni>
    FETCH NEXT FROM Cursor_<Ni> INTO  <VariablesFor_X>, <Variables_PK_u>
    WHILE @@FETCH_STATUS=0
    BEGIN
      IF ExistPRI_<Ni> (<VariablesFor_X>)=0
           <Perform_Activity>
      FETCH NEXT FROM Cursor_<Ni> INTO  <VariablesFor_X>, <Variables_PK_u>
    END
    CLOSE Cursor_<Ni>
    DEALLOCATE Cursor_<Ni>
    FETCH NEXT FROM Cursor_<Nj>  INTO  <VariablesFor_Y>
 END
 CLOSE Cursor_<Nj>
 DEALLOCATE Cursor_<Nj>
```

Figure 7. A parameterized generic pattern of the trigger for the control of tuple deletions

The purpose of the trigger is to check if there is a tuple u in $r(N_i)$ that references only a tuple v in $r(N_j)$, which is marked for the deletion. If it is so, a specified action is initiated. Otherwise, v is deleted from $r(N_j)$, regardless of the specified constraint action.

Since the trigger syntax of MS SQL Server does not include the FOR EACH ROW clause, cascaded cursors are used in the parameterized pattern in Fig. 7. In the process of generating a trigger from the pattern, parameter <N_j> is replaced by the relation scheme name N_j and <N_i> is replaced by the name of N_i. Each constraint has its own name that is embedded into the trigger name by replacing the parameter <Constraint-Name>. <DeclarationFor_Y> and <DeclarationFor_X> represent lists of variable declarations of the form @<Attribute_From_Y> *data type*, and @<Attribute_From_X> *data type*, for each attribute in *Y* and *X*, respectively. <Declaration-For_PK_u> is a list of variable declarations of the form @<Attribute_From_PKey> *data type*, each one for a primary key attribute of N_i. *Deleted* in the statement DECLARE Cursor_<N_j>... is a table with all tuples deleted from $r(N_j)$.

Parameter <VariablesFor_Y> in Fig. 7 is replaced by the list of variables defined by <DeclarationFor_Y>, where each variable is of the form @<Attribute_From_Y>. The variables take values from $v[Y]$, where v is a tuple for deletion. <AttributeSet-From_Y> and <AttributeSetFrom_X> represent the lists of all attributes from *Y* and *X*, respectively. <AttributeSet_PK_u> represent the list of primary key attributes of N_i. <SelectionCriteria_Cursor_N_i> is specified as a sequence of comparison expressions connected by the logical operator AND:

(<Attribute_From _X> IS NULL OR
<Attribute_From _X>=@<Attribute_From _Y>),

where each <Attribute_From _X> or <Attribute_From _Y> belongs to <Attribute-SetFrom_X> or <AttributeSetFrom_Y>, respectively.

Parameter <VariablesFor_X> is replaced by the list of variables defined by <DeclarationFor_X>, where each variable is of the form @<Attribute_From_X>. In the same way, <Variables_PK_u> is replaced by the list of variables defined by <DeclarationFor_PK_u>. These variables take their values from a tuple *u*. Depending on the constraint action selected by the user, parameter <Perform_Activity> is replaced by one of the following procedures: NoAction_<N_j>, SetNullPRI_<N_j>, SetDefault-PRI_<N_j> and CascadeDelPRI_<N_j>. Current primary key values of <Varibales_PK_u> are passed to all of the procedures, except to the first one.

A parameterized pattern of the function ExistPRI_<N_i> is shown in Fig. 8. For each primary key value of a tuple *u*, <SelectionCriteria> is specified as a sequence of comparison expressions connected by the logical operator AND:

(@<Attribute_From _X> IS NULL OR
v.<Attribute_From _Y> = @<Attribute_From _X>).

```
CREATE FUNCTION ExistPRI_<Ni>  (<DeclarationFor_X>)
RETURNS int
AS
BEGIN
  DECLARE @Count int,   @Ret  int
  SELECT @Count = COUNT(*) FROM <Nj>  v WHERE <SelectionCriteria>
  IF @Count != 0 SELECT @ret=1
  ELSE SELECT @ret=0
 RETURN @ret
END
```

Figure 8. A parameterized pattern of function ExistPRI_<N_i>

Procedure NoAction_<N$_j$> is presented in Fig. 9. It is used to implement the constraint action No Action. Procedure SetNullPRI_<N$_j$> is presented in Fig. 10. It is used to implement the constraint action Set Null. <Attribute_value> is a sequence of comma separated expressions, one for each attribute from *X*, specified as follows:

u.<Attribute_From _X> = NULL.

<SelectionCriteria> is a sequence of expressions connected by AND, as follows:

u.<Attribute_From _PK> = @<Attribute_From _PK>.

```
CREATE PROCEDURE NoAction_<Nj>
AS
  RAISERROR('Tuple cannot be deleted from the specified relation ', 16, 1)
  ROLLBACK TRAN
```

Figure 9. A parameterized pattern of the No Action

```
CREATE PROCEDURE SetNullPRI_<Nj>  (<DeclarationFor_PK_u>)
AS
  UPDATE u SET <Attribute_value> FROM <Ni> u WHERE <SelectionCriteria>
```

Figure 10. A parameterized pattern of the Set Null action

Procedure SetDefaultPRI_<N$_j$> used to implement Set Default action is presented in Fig. 11. <ValueAssignmentFrom_X> is replaced as follows:

@<Attribute_From _X>= *u*.<Attribute_From _X>.

```
CREATE PROCEDURE SetDefaultPRI_<Nj> (<DeclarationFor_PK_u>)
AS
  DECLARE <DeclarationFor_X>, @AttributesForUpd VARCHAR(255)
  SET @ AttributesForUpd  = ' '
  SELECT <ValueAssignmentFrom_X> FROM <Ni> u WHERE <SelectionCriteria>
  IF (<UpdateCondition>)
  BEGIN
   IF (@<Attribute_From _X> IS NOT NULL)
   BEGIN
     IF @AttributesForUpd != ''
       SET  @AttributesForUpd=@AttributesForUpd+',
       u.<attribute_from_X> = default'
     ELSE
        SET  @AttributesForUpd = 'u.<attribute_from_X> = default'
   END
   EXEC ('UPDATE u SET ' + @AttributesForUpd + 'FROM <Ni>  u WHERE
        u.<Attribute_From_PK>=' +@<Attribute_From_PK>)
   SELECT <ValueAssignmentFrom_X> FROM <Ni> u WHERE
       <SelectionCriteria>
   IF dbo.ExistPRI_<Ni>(<VariablesFor_X>)=0
   BEGIN
     RAISERROR('Tuple cannot be deleted from the specified relation ', 16, 1)
     ROLLBACK TRAN
   END
  END
```

Figure 11. A parameterized pattern of the Set Default action

<SelectionCriteria> is a sequence of expressions connected by AND, one for each attribute from <Attribute_From_PK>, specified as follows:

(*u*.<Attribute_From _PK> = @<Attribute_From _PK>).

Since only the attributes having non null values are set to the default values, the first IF statement in Fig. 11 checks if there is at least one having a non null value. <UpdateCondition> is a sequence of the expressions connected by OR, one for each attribute in *X*, specified as follows:

@<Attribute_From _X> IS NOT NULL.

The bolded code in Fig. 11 is repeatedly generated, once for each attribute in *X*. Therefore, for each attribute in *X* having a non null value, a string

'*u*.<Attribute_From_X> = default'

is concatenated to the current value of the variable @AttributesForUpd.

The WHERE clause of the UPDATE command in the string used in EXEC command is correctly defined in Fig. 11, if the primary key of N_i consists of the only one attribute. Otherwise, the clause is transformed to include an expression of the form

u.<Attribute_From_PK> = @<Attribute_From_PK>,

for each primary key attribute, and all such expressions are connected by the AND operator.

Procedure CascadeDelPRI_<N_j> used to implement Cascade action is presented in Fig. 12. <SelectionCriteria> is an expression of the form <SelectionCriteria1> AND <SelectionCriteria2>. <SelectionCriteria1> is a sequence of expressions connected by AND, one for each attribute in <Attribute_From_PK> specified as:

(u.<Attribute_From_PK> = @<Attribute_From_PK>).

<SelectionCriteria2> is a sequence of expressions connected by AND, one for each attribute in *X*, specified as:

(u.<Attribute_From_X> IS NOT NULL).

```
CREATE PROCEDURE CascadeDelPRI_<Nj>  (<DeclarationFor_PK_u>)
AS
   DELETE FROM <Ni> u  WHERE <SelectionCriteria>
```

Figure 12. Parameterized pattern of procedure for cascade delete

In this example, we have presented one of the cases met in the real life. We have deliberately selected here the partial referential integrity constraint, since its implementation is the most complex. Also, we have deliberately selected MS SQL Server, because it makes the implementation even more complex, since its trigger syntax does not support FOR EACH ROW clause.

5 Conclusion

The paper describes SQL Generator that is a component of IIS*Case. IIS*Case is a complex software tool that supports automatic generation of 3NF db schemas and software applications. In the framework of IIS*Case, SQL Generator provides users with such an intelligent support that they can generate implementation specifications of db schemas even without knowing the SQL syntax and procedural DBMS mechanisms for the implementation of constraints.

An advantage of SQL Generator over other similar products is that users have a wider selection of possible actions to preserve db consistency. Besides the generation of common db constraints, like key, unique, not null, and native referential integrity, SQL Generator also enables the implementation of the default, partial and full referential integrity constraints, and the selection of an appropriate action from the set {No Action, Cascade, Set Default, Set Null}. Also, SQL Generator provides the implementation of the inverse referential integrity constraints. SQL Generator validates selections of input parameter values, analyzes designer's solutions, and issues warnings if it detects any inconsistency.

Further research and development are focused on extending the functionality of SQL Generator. We plan to:

- implement the generation of the extended referential integrity [9],
- provide the compatibility checking of data types,
- add modules for the design and implementation of physical data structures for particular DBMSs,
- implement in IIS*Case visual editors for specifying user defined functions and tuple (check) constraints [8], and
- enable generating SQL scripts for a wider selection of DBMSs.

References

1. Aleksić S., An SQL Generator of Database Schema Implementation Specification in a CASE Toll IIS*Case, M.Sc. (Mr.) Dissertation, University of Novi Sad, Faculty of Technical Sciences, Novi Sad, Serbia, 2006;
2. ARTech. *DeKlarit*™ (*The Model-Driven Tool for Microsoft Visual Studio 2005*), Chicago, U.S.A. Available at: http://www.deklarit.com [June, 2007].
3. Govedarica M., Design the Set of Implementational Database Schema Constraints, M.Sc. (Mr.) Dissertation, University of Novi Sad, Faculty of Technical Sciences, Novi Sad, 1998;
4. ISO/IEC 9075-{1, 2, 11}:2003 (ANSI SQL:2003), American National Standards Institute;
5. Luković I, Mogin P, Pavićević J, Ristic S, An Approach to Developing Complex Database Schemas Using Form Types, Software: Practice and Experience, John Wiley & Sons Inc, Hoboken, USA, ISSN: 0038-0644, Published Online, May 29, 2007, DOI: 10.1002/spe.820;
6. Microsoft SQL Server 2000, User Manuals;
7. Microsoft SQL Server 2005, User Manuals;
8. Mogin P, Luković I, Govedarica M, Database Design Principles, 2nd Edition, University of Novi Sad, Faculty of Technical Sciences, Novi Sad, Serbia, 2004, ISBN: 86-80249-81-5;
9. Mogin P, Luković I, Govedarica M, Extended Referential Integrity, Novi Sad Journal of Mathematics, Novi Sad, Serbia, ISSN: 1450-5444, Vol. 30, No. 3, 2000, pp. 111-122.
10. Oracle DBMS 9i, User Manuals;
11. Oracle DBMS 10g, User Manuals;
12. Oracle Designer 9i, On-line Documentation;
13. Pavićević J, Development of A CASE Tool for Automated Design and Integration of Database Schemas, M.Sc. (Mr.) Dissertation, University of Montenegro, Faculty of Science, Podgorica, Montenegro, 2005;
14. Pavićević J, Luković I, Mogin P, Govedarica M, Information System Design and Prototyping Using Form Types, INSTICC I International Conference on Software and Data Technologies, Setubal, Portugal, September 11-14, 2006, Proceedings, Vol. 2, pp. 157-160;
15. Sybase PowerDesigner 10, On-line Documentation.