Microsoft's new database modeling tool: Part 6

Terry Halpin
Microsoft Corporation

Abstract: This is the sixth in a series of articles introducing the Visio-based database modeling component of Microsoft Visual Studio .NET Enterprise Architect. Part 1 showed how to create a basic ORM source model, map it to a logical database model, and generate a DDL script for the physical database schema. Part 2 discussed how to use the verbalizer, make an object type independent, objectify an association, and add some other ORM constraints to an ORM model. Part 3 showed how to add set-comparison constraints (subset, equality and exclusion) and how exclusive-or constraints combine exclusion and disjunctive mandatory constraints. Part 4 discussed the basics of modeling and mapping subtypes. Part 5 discussed mapping subtypes to separate tables, and occurrence frequency constraints. Part 6 discusses ring constraints.

Introduction

This is the sixth in a series of articles introducing the database modeling solution in Microsoft Visio for Enterprise Architects, which is included in the Enterprise Architect edition of Visual Studio. NET. This article discusses how to add ring constraints to an ORM model, and how these constraints are enforced in the resulting relational schema. Familiarity with ORM and relational database modeling is assumed. For an overview of ORM, see [1]. For a thorough treatment of ORM and database modeling, see [2]. For previous articles in this series, see [3], [4], [5], [6] and [7].

Ring constraints

In one university I lectured at some years ago, it was common practice for an academic to be monitored by another academic. Monitoring involved attending a lecture by the other academic, and giving him/her feedback on how to improve it. Although each academic had at most one monitor, it was possible for the same academic to monitor many other academics. This monitoring relationship is modeled as an n:1 binary fact type in the simple ORM model shown in Figure 1.

Notice that both roles in the is-monitored-by predicate are played by the same object type, Academic. The fact type path goes from the object type through the role pair and back to the object type, forming a ring. For this reason, the fact type Academic is monitored by Academic is called a *ring fact type*. Since both roles are played by the same object type, it is meaningful to compare instances in their populations, and the roles are said to be *compatible*. In practice, most ring fact types need further constraints on how their role instances may be logically related. Such constraints are called *ring constraints*. See if you can think of some constraint that needs to be added to the monitor fact type, before reading on.

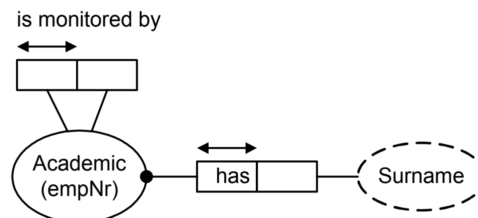

Figure 1 The monitoring association is a ring fact type

As you may have guessed, we should at least require that no academic is monitored by himself/herself. Technically this is said to be an *irreflexive* ring constraint, and is denoted by the symbol "^{O}*ir*" besides the two roles involved, as shown in Figure 2. The "^{O}" suggests a ring, and the "*ir*" is short for "irreflexive".

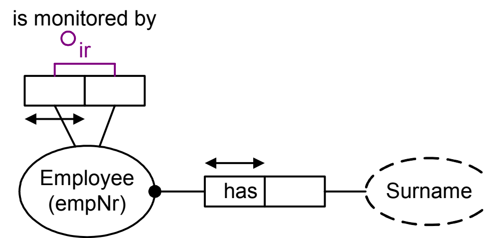

Figure 2 The irreflexive ring constraint ensures that no academic monitors himself/herself.

Let's see how to add this constraint using Visio for Enterprise Architects. First create the ORM source model shown in Figure 1, using techniques described in earlier articles. Now select the monitor predicate, right-click it, and select Add Constraints... from the menu, to bring up the Add Constraint dialog. Now choose Ring from the Constraint type field, and click the two roles in the displayed predicate (see Figure 3).

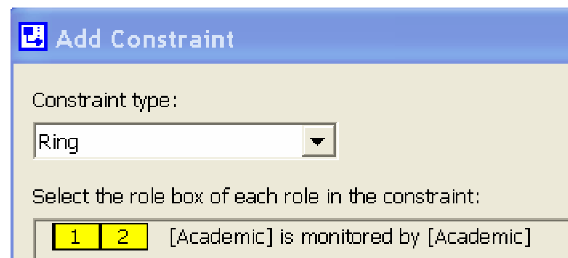

Figure 3 Adding a ring constraint using the Add Constraint dialog

As you select the two roles, they are automatically numbered 1 and 2, and the Ring Constraint Properties dialog appears, with the mouse cursor appearing as a cross-hair. Move this cursor to select the ellipse marked ir (irreflexive), which sets the constraint's Ring Type to Irreflexive (see Figure 4).

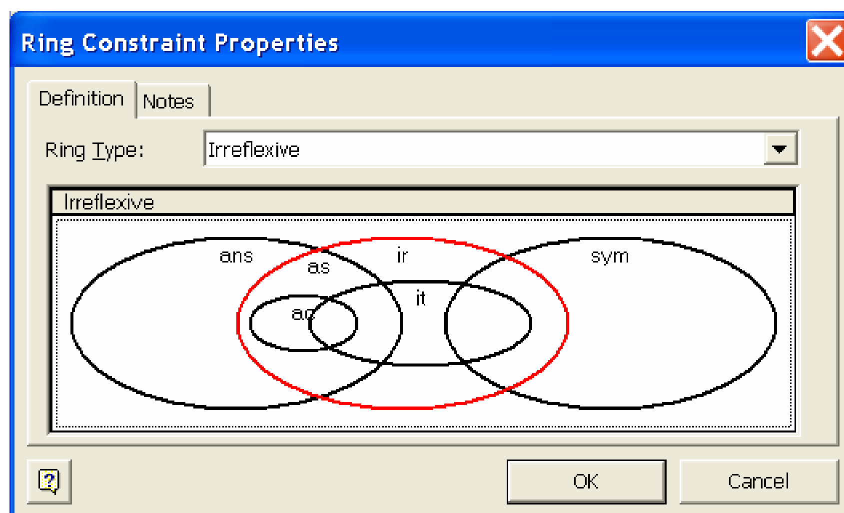

Figure 4 Choosing the specific ring constraint type to be irreflexive

When you hit the OK button, the Ring Constraint Properties dialog disappears, and the Add Constraint dialog now displays the constraint verbalization as “No Academic is monitored by itself”, as shown in Figure 5. If you would rather see the verbalization at the same time as you move the cross-hair cursor over the constraint selector, you can alternatively enter ring constraints directly via the fact editor.

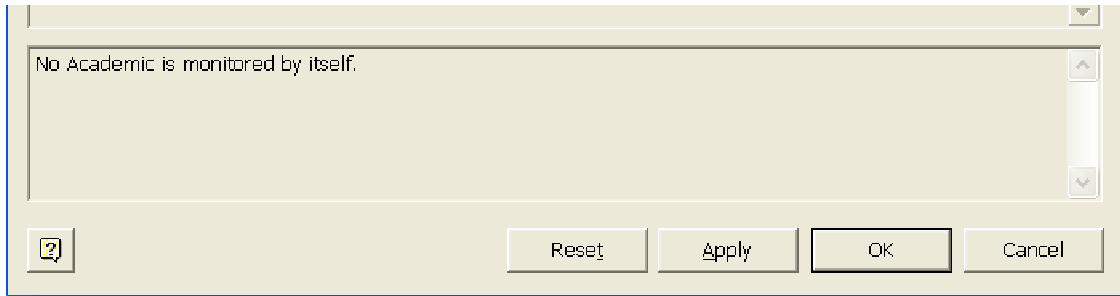

Figure 5 Verbalization of the irreflexive constraint

When you hit the OK button, the dialog disappears and the ring constraint is added to the diagram, as shown in Figure 2. You can finesse the diagram by repositioning things as I’ve done here using standard Visio controls (use Flip Vertical on the predicate to move its uniqueness constraint to the other side, then select and drag the predicate text). As usual, you can see the verbalization of all constraints on the predicate by selecting it and opening the Verbalizer.

A *ring constraint* may apply only to a pair of roles played by the same (or a compatible) object type. The role pair may form a binary predicate or be embedded in a longer predicate. Let *R* be the relation type comprising the role pair. Using “iff” for “if and only if”, “~” for “not” and “→” for “implies”, we say that *R* is *irreflexive* iff for all *x*, *~xRx*. This is denoted by connecting the role pair to the irreflexive constraint symbol “^{O}*ir*”.

As shown in Figure 4, the tool allows you to specify six kinds of ring constraint. The other five may be summarized as follows. *R* is *symmetric* iff for all *x, y*, *xRy* → *yRx*. Symmetry is used more often for derivation, but if used as a constraint, it is denoted by connecting the role pair to “^{O}*sym*”. *R* is *asymmetric* (^{O}*as*) iff for all *x, y*, *xRy* → *~yRx*. *R* is *antisymmetric* (^{O}*ans*) iff for all *x, y*, *x* ≠ *y* & *xRy* → *~yRx*. *R* is *intransitive* (^{O}*it*) iff for all *x, y, z, xRy* & *yRz* → *~xRz*. Asymmetry and intransitivity each imply irreflexivity. Exclusion implies asymmetry (and irreflexivity). An irreflexive, functional relation must be intransitive. One very expensive ring constraint to enforce is *acyclicity* (^{O}*ac*), i.e. no cycles are permitted by using the relationship type one or more times.

Although the Euler diagram in Figure 4 may appear complex, I designed it to simplify things for you, so that you wouldn’t need to worry about avoiding incompatible or redundant ring constraints. For example, if you choose Asymmetric, the tool won’t let you choose Irreflexive as well, because this is implied by asymmetry. For the monitoring relationship discussed earlier, it is possible for two academics to monitor each other, so this is a simple case of irreflexivity, not asymmetry.

With many fact types, it’s often a good idea to include at least one role name in addition to the predicate name(s), since this enables better control over how column names are generated for the logical database model. This is especially true for ring fact types. For example, to add the role name “monitor” to the second role of Academic is monitored by Academic, select the fact type to open up its Database Properties window, then open the Readings pane, and enter “monitor” for the name of the second role, as shown in Figure 6.

Although the tool does not display role names on the diagram, you can always add them yourself manually using a text box. If you do this, I suggest you enclose the role names in square brackets to distinguish them from other names, as shown in Figure 7.

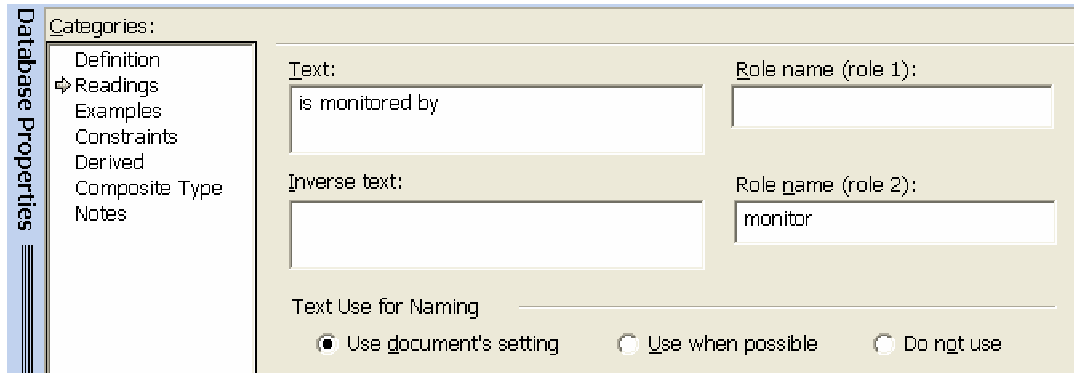

Figure 6 Adding the name “monitor” for the second role of “Academic is monitored by Academic”

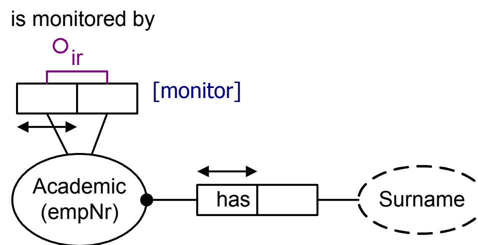

Figure 7 If desired, role names may be added manually to diagrams using text boxes

To map the ORM model to a logical database model, first save your ORM model, then open a new database model (File > New > Database > Database Model Diagram), add the ORM source model to it (Database > Project > Add Existing Document…), and then build the database model (Database > Project > Build), saving it when prompted. This results in a relational database model with only one table scheme. When you drag it onto the drawing window, it should look like the one shown in Figure 8.

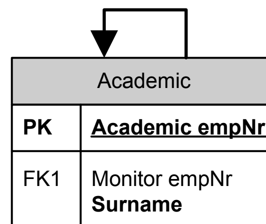

Figure 8 The logical database model mapped from the ORM model in Figure 7

If you wish to finesse the logical model by renaming or moving columns, select the table to open its Database Properties Sheet, then choose the Columns category, edit the column names as desired, and use the Move Up button to move a column upwards (see Figure 9). Later articles will discuss how to get finer control over column name generation, and how to use various features in the logical database model solution.

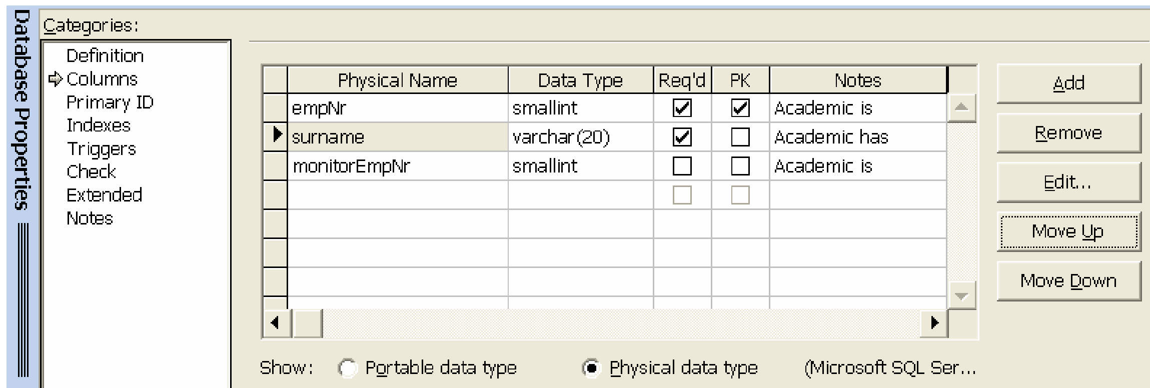

Figure 9 Renaming and moving columns

After making the changes shown, the relational model should appear as shown in Figure 10. Hit the Save icon to save the model in this form. This will bring up a dialog asking whether you want to migrate those changes back to source models (in this case the only source model is our ORM model). Answer No to this prompt. Despite the rather threatening phrasing of the migrate dialog box, there is rarely any need to migrate non-structural changes such as renaming columns or moving non-key columns, unless you want to reuse the source models in other projects and avoid such minor reformatting.

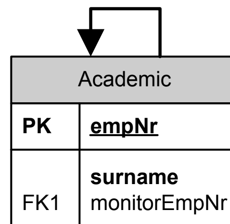

Figure 10 The table scheme after renaming and moving columns

Now let's generate the physical database schema and look at the DDL. To do this from the database model solution, choose Database > Generate…, accept the default setting to generate just the DDL, choose your database driver (e.g. SQL Server 2000), enter a database name (e.g. "RingDB"), accept the defaults and answer Yes to view the DDL script. The main aspects of the resulting DDL are shown. Notice how the irreflexive ring constraint was enforced using the check condition `empNr <> monitorEmpNr`.

```
create table "Academic" (
        "empNr" smallint not null,
        "surname" varchar(20) not null,
        "monitorEmpNr" smallint null)

/* Create table level checks for table Academic.                                   */
alter table "Academic" add constraint Academic_ring check ( "empNr" <> "monitorEmpNr" )

alter table "Academic"
        add constraint "Academic_PK" primary key ("empNr")

/* Add foreign key constraints to table "Academic".                                 */
alter table "Academic"
        add constraint "Academic_Academic_FK1" foreign key (
                "monitorEmpNr")
         references "Academic" (
                "empNr") on update no action on delete no action
```

It is possible that more than one atomic ring constraint applies to the same predicate. For example, the fact type Person is parent of Person in Figure 11 is declared to be both asymmetric (as) and intransitive (it). The tool treats this as a single, composite ring constraint, as indicated by bracketing the two components (as,it). To add this constraint, use the Add Constraint dialog as before, and position the cross-hair cursor to obtain the “Asymmetric + Intransitive” option. To add the “<=2” frequency constraint that each child has at most two parents, use the Add Constraint dialog, select frequency for the constraint type, select the second role, and accept the default minimum and maximum values (see previous article for more details on frequency constraints).

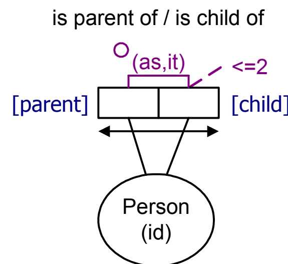

Figure 11 The parenthood fact type is asymmetric and intransitive

To facilitate checking of incompatible or duplicate ring constraints, the tool relies on the Euler diagram interface for ring constraints, allowing only one ring constraint (simple or composite) per predicate. So if you previously added one ring constraint to a predicate, and wish to add a second one to it, you need to edit the original constraint, replacing it with the composite constraint. To edit an existing constraint on a single predicate, select the predicate to invoke its Database Properties sheet, choose the Constraints category, select the constraint, and then hit the Edit button.

The tool verbalizes the asymmetric and intransitive constraints respectively thus:

If Person p1 is parent of Person p2
then it cannot be that
Person p2 is parent of Person p1.

If Person p1 is parent of Person p2
and Person p2 is parent of Person p3
then it cannot be that
Person p1 is parent of Person p3.

The intransitive constraint assumes that parenthood is restricted to genetic parenthood, and that no incest is allowed. Before mapping the parenthood fact type, it’s a good idea to enter the role names “parent” and “child” for its first and second roles. I’ve displayed these role names on the ORM diagram in text boxes. To map the parenthood fact type, add it to a database model and build it in the usual way. This results (after some minor renaming) in the all-key table scheme shown in Figure 12.

Parenthood	
PK	**parentId**
PK	**childId**

Figure 12 The relational model for parenthood (some constraints are not displayed)

To generate the physical database schema from the database model solution, choose Database > Generate…, accept the default setting to generate just the DDL, choose your database driver, enter a database name, accept the defaults and answer Yes to view the DDL script. The main aspects of the resulting DDL (choosing SQL Server for the target DBMS) are shown.

```
create table "Parenthood" (
        "parentId" smallint not null,
        "childId" smallint not null)

alter table "Parenthood"
        add constraint "Parenthood_PK" primary key ("parentId", "childId")

/* Create procedure/function Parenthood_freq1.                                   */
/* /* The constraint:                                                       */ */
/* /* Each Person p that occurs in                                          */ */
/* /*    Person is parent of Person p                                       */ */
/* /*     occurs there at most 2 times.                                     */ */
/* /* is enforced by the following DDL.                                     */ */
Create Procedure sp_Parenthood_freq1 as
/*   Microsoft Visual Studio generated procedure code. */
if (
    not exists (select * from "Parenthood"
            group by "Parenthood"."childId"
            having count(*) > 2)
)
  return 1
else
  return 2
/* End sp_Parenthood_freq1                                             */


/* Create procedure/function Parenthood_ring2.                                   */
/* /* The constraint:                                                       */ */
/* /* If Person p1 is parent of Person p2                                   */ */
/* /*    and Person p2 is parent of Person p3                               */ */
/* /*    then it cannot be that                                             */ */
/* /*    Person p1 is parent of Person p3.                                  */ */
/* /* If Person p1 is parent of Person p2                                   */ */
/* /*    then it cannot be that                                             */ */
/* /*    Person p2 is parent of Person p1.                                  */ */
/* /* is enforced by the following DDL.                                     */ */
Create Procedure sp_Parenthood_ring2 as
/*   Microsoft Visual Studio generated procedure code. */
if (
    not exists (select * from "Parenthood" X, "Parenthood" Y
            where X."parentId" = Y."childId" and
                X."childId" = Y."parentId")

and
    not exists (select * from "Parenthood" X, "Parenthood" Y, "Parenthood" Z
            where X."childId" = Y."parentId" and
                Y."childId" = Z."childId" and
                X."parentId" = Z."parentId")
)
  return 1
else
  return 2
/* End sp_Parenthood_ring2
```

The irreflexive constraint discussed earlier was enforced as a simple check clause, since its condition for any given row could be checked by looking at that row alone. This is not true for the frequency constraint and the asymmetric and intransitive ring constraints. Instead of a check clause, the tool generates stored procedures for these more complex constraints. If a procedure returns 1, then the constraint is satisfied; if it returns 2, the constraint is violated. These procedures could be expensive, so are not run automatically. It is your responsibility to decide how to use them. For example, you might write code to run the procedures, or you might replace the procedures by equivalent triggers.

The procedural code is commented, and should be self-explanatory. Notice that the composite (as,it) ring constraint maps to a single stored procedure, with the first existential subquery checking asymmetry, and the second existential subquery checking intransitivity. For those target DBMSs that do not support stored procedures (e.g. Microsoft Access), the stored procedures are documented as comments, which you can use to help develop an alternative way to enforce the constraints.

In reality, the parenthood relation is not just asymmetric, but acyclic. Acyclicity is different from the other ring constraints, because it involves recursion, and hence might be quite expensive to enforce. Some modern DBMSs, such as SQL Server, now include the capability of performing recursive queries, but not all DBMSs do so. Moreover, the SQL syntax used to support recursion may differ among DBMSs. If you wish to declare an acyclic ring constraint, the tool currently does no more than generate a comment in the DDL for it, so you have to look after coding this yourself. Moreover, if you choose a composite ring constraint that includes acyclicity, the whole code for the composite constraint is simply a comment. Although we hope to add full support for acyclic constraints in a later version, this feature has low priority in comparison with robustness improvements and other enhancements currently under consideration.

Conclusion

This article discussed how to specify ring constraints in an ORM model, and how these constraints are enforced in a relational schema. For a more detailed discussion of ring constraints, with lots of simple examples, see section 7.3 of [2]. The next article will discuss index constraints, constraint layers, data types, and ways to control the generation of various aspects (e.g. column names) in the resulting relational model. If you have any constructive feedback on this article, please e-mail me at: TerryHa@microsoft.com.

References

1. Halpin, T. A. 1998 (revised 2001), 'Object Role Modeling: an overview', white paper, (online at www.orm.net).
2. Halpin, T.A. 2001a, *Information Modeling and relational Databases*, Morgan Kaufmann Publishers, San Francisco (www.mkp.com/books_catalog/catalog.asp?ISBN=1-55860-672-6).
3. Halpin, T.A. 2001b, 'Microsoft's new database modeling tool: Part 1', *Journal of Conceptual Modeling*, June 2001 issue (online at www.InConcept.com and www.orm.net).
4. Halpin, T.A. 2001c, 'Microsoft's new database modeling tool: Part 2', *Journal of Conceptual Modeling*, August 2001 issue, (online at www.InConcept.com and www.orm.net).
5. Halpin, T.A. 2001d, 'Microsoft's new database modeling tool: Part 3', *Journal of Conceptual Modeling*, October 2001 issue, (online at www.InConcept.com and www.orm.net).
6. Halpin, T.A. 2002a, 'Microsoft's new database modeling tool: Part 4' (online at www.orm.net). This is a revised version of an earlier article of the same title in the January 2002 issue of *Journal of Conceptual Modeling*.
7. Halpin, T.A. 2002b, 'Microsoft's new database modeling tool: Part 5' (online at www.orm.net). This is a revised version of an earlier article of the same title in the March 2002 issue of *Journal of Conceptual Modeling*.

Note: Revised versions of many of the above references are also accessible online from the MSDN library (http://msdn.microsoft.com/library/default.asp). From the tree browser on the MSDN Library Home Page choose the following path to find these articles: Visual Tools and Languages > Visual Studio .NET > Visual Studio .NET (General) > Technical Articles.