Microsoft's new database modeling tool: Part 7

Terry Halpin
Microsoft Corporation

Abstract: This is the seventh 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 discussed ring constraints. Part 7 discusses index constraints, constraint layers, and data types.

Introduction

This is the seventh 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 index constraints to an ORM model, show/hide constraints via layers, and specify data types. 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], [7] and [8].

Index constraints

Consider the ORM schema shown in Figure 1(a). Here each person is identified by a social security number (ssn), has a surname, at most two given names, and was born in at most one country. Although rare, it is possible that a person has no given names, so the given name fact types are optional. In real life, each person was born in a country, but the information system makes it optional to record a person's birth country. Each country is identified by a country code (e.g. 'AU'), but also has a unique, identifying name (e.g. 'Australia').

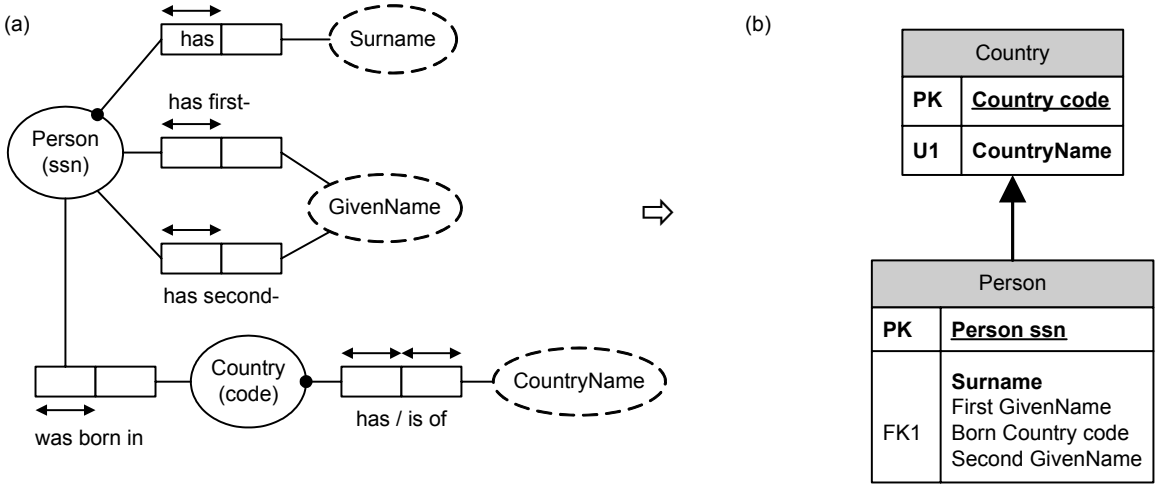

Figure 1 Default mapping of a simple ORM schema

Previous articles discussed how to enter an ORM conceptual schema, map it to a logical database schema, and generate the DDL script for the physical database schema. By default, the sample ORM schema maps to the relational schema shown in Figure 1(b). The "PK" annotation indicates primary keys, "FK*n*" indicates foreign keys, and "U*n*" indicates uniqueness constraints. If you choose SQL Server 2000 as the target system, and generate the DDL script for this schema, the following DDL code is generated (I've removed the generated comments to save space).

```
create table "Person" (
        "Person ssn" char(10) not null,
        "Surname" char(10) not null,
        "First GivenName" char(10) null,
        "Born Country code" char(10) null,
        "Second GivenName" char(10) null)

alter table "Person"
        add constraint "Person_PK" primary key ("Person ssn")

create table "Country" (
        "Country code" char(10) not null,
        "CountryName" char(10) not null)

alter table "Country"
        add constraint "Country_PK" primary key ("Country code")

create unique index "Country_AK1" on "Country" ("CountryName")

alter table "Country" add constraint "Country_AK1_UC1" unique ("CountryName")

alter table "Person"
        add constraint "Country_Person_FK1" foreign key ("Born Country code")
         references "Country" ("Country code")
```

Notice that data types are all char(10) by default. We'll see how to change this in the next section. For now, let's focus on constraints that have a bearing on *indexes*. Just as an index to a book enables you to quickly find a topic of interest, indexes on database columns enable the database system to quickly access entries for those columns. While indexes speed up access (e.g. using binary-trees in RAM), they can also slow down updates (because the index also needs to be updated), so care is required in choosing them. Optimizing performance via index selection is a large topic (e.g. see chapter 31 of [9]), especially with the many kinds of indexes available nowadays, so we restrict our discussion here to the basics.

Like most DBMSs, SQL Server automatically creates unique indexes on primary keys, since this provides an efficient way to enforce uniqueness constraints. Moreover, primary key columns are often involved in join and sort operations, so require efficient access. Since the primary key declarations in the DDL script automatically cause SQL Server to create indexes for them, there is no need to explicitly declare indexes on PK columns.

As Figure 1(b) indicates, the CountryName column is mandatory and unique, and hence provides an alternate key for Country. The DDL script above includes a declaration to add a unique constraint (Country_AK1_UC1) to this column. SQL Server automatically creates unique indexes on columns for which unique constraints are declared, since this provides an efficient way to enforce the constraint. So there is no need to explicitly declare a unique index for this column. The above DDL script however, which was generated from the SR1 release of the modeling tool, redundantly includes a create-unique-index statement for this column.

This redundant behavior has been fixed for the next release, but in the meantime you can fix this yourself. To do this, double-click the Country table scheme to bring up its Database Properties window, select the Indexes category, open the drop-down list for Index type, and select the Unique constraint only option (instead of Unique index with constraint on top), as shown in Figure 2. Now save your change, and migrate it back to the ORM source model when prompted.

If you regenerate the DDL, you will see that uniqueness on Country.CountryName is enforced by a unique constraint (Country_UC1) without an additional unique index clause.

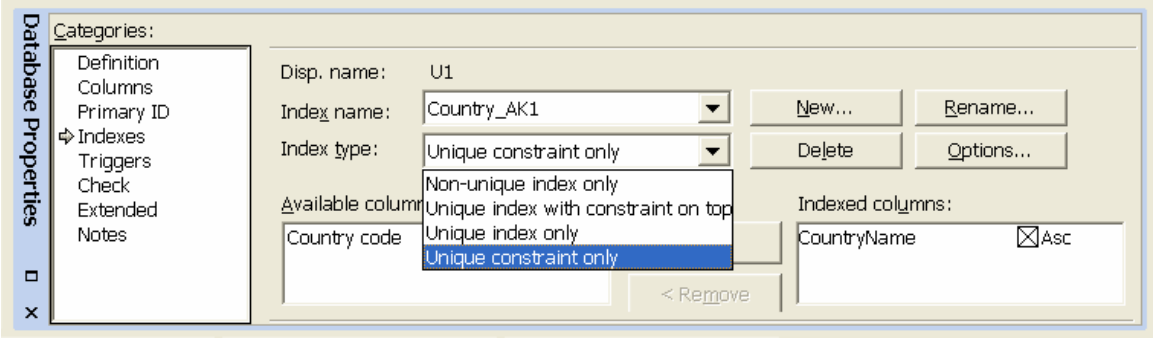

Figure 2 Choosing to implement uniqueness with a declarative unique constraint

Strictly speaking, indexes belong to the physical level. However, it is possible to specify indexes directly on an ORM source model. This enables you to control the mapping process up front by annotating the pure conceptual model with implementation detail. Let's see how to do this.

Suppose we often want to access personal details without having to specify the person's social security number, but rather by entering a surname to list all employees with that surname. To make this access more efficient we could declare an index on the role played by Surname in the ORM model of Figure 1(a). To do this, select the Person has Surname predicate, right-click it to bring up its context menu, then select "Add Constraints..." to bring up the Add Constraint dialog. Now choose Index from the Constraint type drop-down list, select the right-hand role, as shown in Figure 3.

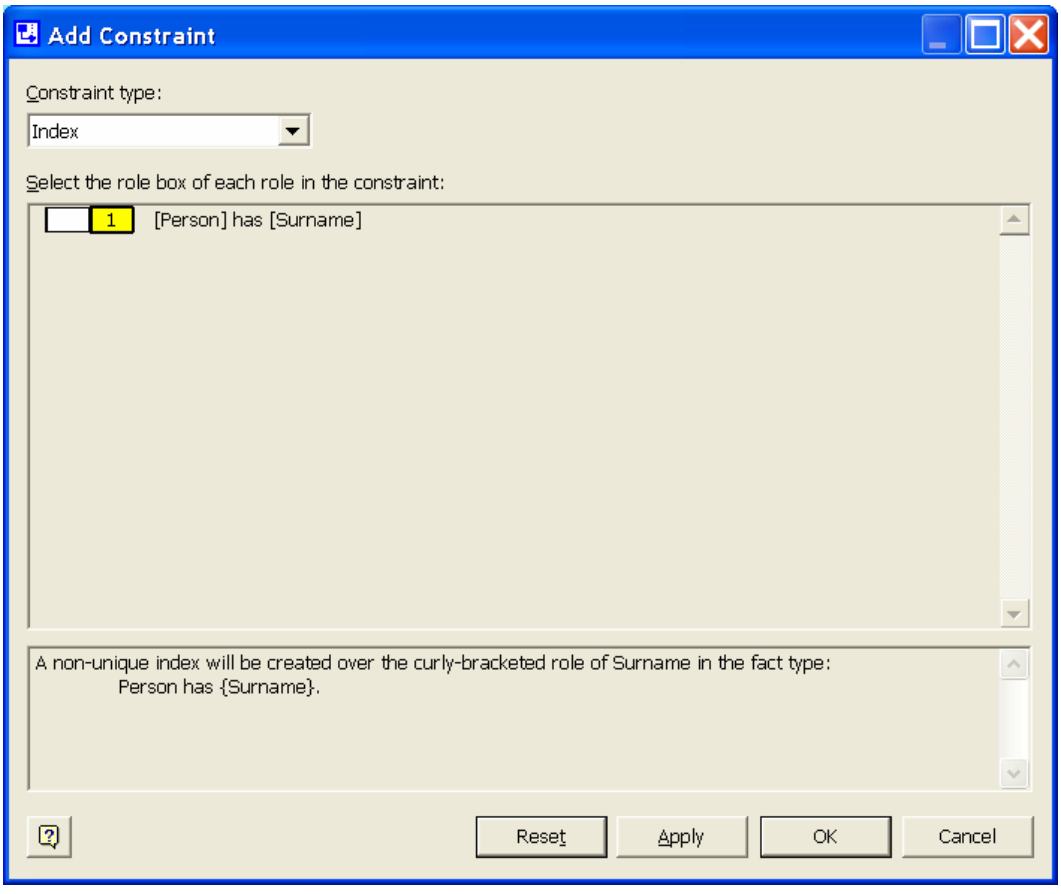

Figure 3 Adding an "index constraint" to the role played by Surname

The verbalization indicates that a non-unique index will be created over the role played by Surname. This ensures that when the model is mapped to a relational schema, a non-unique index will be created over the column(s) to which that role maps. Since many people may have the same surname, the index will be non-unique. Hit the OK button to apply the index and exit the dialog. An "index constraint" now appears as a circled "I" on the ORM model, attached to the surname role, as shown at the top of Figure 4. Although stored with the ORM model, this physical annotation is not part of the pure conceptual schema. To assist with our discussion of constraint layers in the next section, where this diagram will be referenced, I've added a subset constraint as well. Ignore this for now.

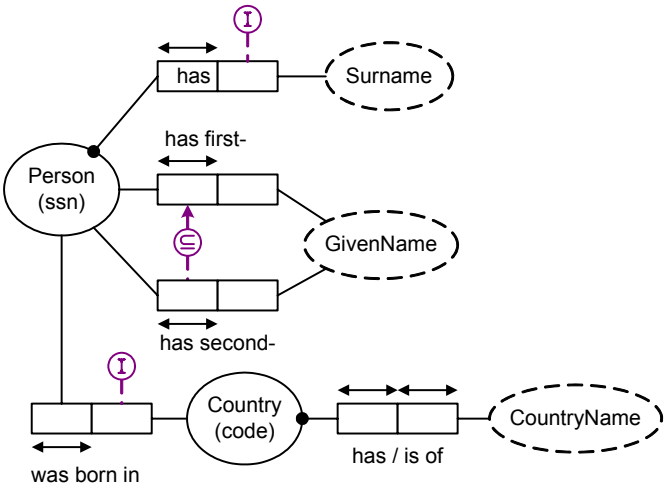

Figure 4 An "index constraint" appears as a circled "I" attached to the role(s) it constraints

Another candidate for an index is the role played by Country in the association Person was born in Country. If you often want to inquire about where a person was born, and you want the name of the country, not just the country-code, when you make such a query, then this entails a frequent conceptual join operation between the two roles played by Country. These roles map to separate tables, one role mapping to a foreign key and the other to its referenced primary key. At the relational level, this means we have a need for efficiently joining the tables by matching the FK and PK values. SQL Server does not automatically generate indexes for foreign keys, so if a foreign key is frequently needed for a join you should consider adding an index for it. In this case, you can add an index on the birth-country role by clicking on its fact type and using the Add Constraint dialog (as described earlier for the surname role). The resulting index constraint is shown at the bottom of Figure 4.

Constraint layers

One of ORM's strengths is its richly expressive constraint notation. When specifying detailed requirements, it often helps a great deal if we can visualize the relevant business rules graphically. However, there may also be times when we wish to ignore this finer level of detail. One way of doing this in ORM is to use *constraint layers* to control whether various kinds of constraints are displayed on screen and/or printed. In Microsoft Visio for Enterprise Architects, constraint layers are implemented using Visio's standard layer properties mechanism.

For example, the ORM model in Figure 4 displays four kinds of constraints: simple mandatory; internal uniqueness, subset, and index. The subset constraint (circled "⊆") indicates that if a person has a second given name then that person also has a first given name. As described in [5], subset constraints can be added by selecting their predicates, right-clicking to invoke the Add Constraints dialog, and making relevant choices. Because of their fundamental nature, simple mandatory and internal uniqueness constraints are always displayed. But all other kinds of ORM constraints can have their display suppressed by controlling the layer properties settings for the diagram. To access the Layer Properties dialog, choose View > Layer Properties... from the main menu. The dialog for our current model is shown Figure 5.

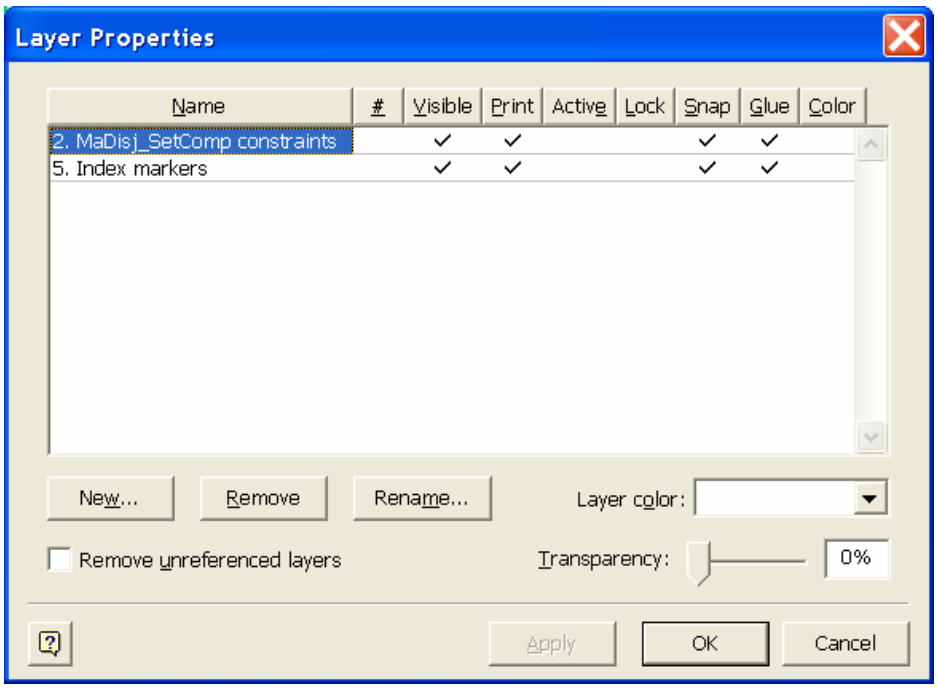

Figure 5 Constraint layers allow most kinds of constraints to be hidden

Different kinds of constraint exist on different layers. In addition to the always-shown constraints (simple mandatory and internal uniqueness), this model includes constraints on layers 2 and 5. Layer 2 includes disjunctive mandatory (inclusive-or) and set-comparison (subset, equality, exclusion) constraints. Layer 5 includes index markers (a better term for index constraints). To suppress display on screen of a constraint layer, click the check mark in the Visible column in this dialog (this removes the check mark), then hit the Apply button. If you remove both check marks for Visible in this example, Figure 4 will be redisplayed as Figure 1(a). The diagram will still print as Figure 4 unless you uncheck the entries in the Print column. The column entries are toggles, so you can restore a check mark to a cell simply by clicking it. ORM models may include other kinds of constraint that exist on other layers.

With the exception of simple mandatory and internal uniqueness constraints, ORM constraints are partitioned into five layers, as shown in Figure 6. Each layer can be individually controlled. Layer 1 includes external uniqueness constraints. Layer 2 includes disjunctive-mandatory (inclusive-or) and set comparison (subset, equality, exclusion) constraints. Since exclusive-or constraints are simply combinations of inclusive-or and exclusion constraints, these are included on layer 2. Layer 3 holds value constraints. Layer 4 displays frequency and ring constraints. Finally, layer 5 is used for indexes. By default, all constraints are displayed and printed, unless their display/print setting is unchecked.

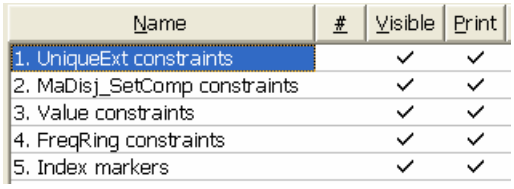

Figure 6 Five layers of ORM constraints may be suppressed

Data types

Although data types for columns may be specified at the relational level, it is far better to specify *data types* at the ORM level. Why? Firstly, it saves a lot of work, because data types in ORM correspond to the syntactic *domains* on which relational attributes are based. Typically each ORM object type plays many roles, each of which maps to one or more columns in a relational database. Setting the data type once for the object type propagates that data type to every attribute mapped from its roles. Secondly, this avoids type mismatch problems in the generated relational database (e.g. foreign keys are automatically given the same data type as the primary keys they reference).

Thirdly, it makes it much easier to change data types. For example, a database might include hundreds of columns concerning dates (birthdate, hiredate, orderdate etc.). Suppose you need to change the data type for each of these columns from, say char(10) where dates were stored as character strings like '2002-07-06', to a datetime data type that has built-in support for date arithmetic. At the ORM level, all you need do is change the data type for the object type Date. At the relational level, you need to change all the hundreds of date column data types (unless your DBMS properly supports relational domains, and you have been disciplined in using this support). Of course, after changing the schema for a populated database you still have the data migration problem, but having the schema updated so easily is a major benefit.

By default, each object type in an ORM model is assigned a data type of char(10). Before mapping an ORM model to a relational model, the relevant data types should be specified. The tool offers more than one way to do this. One way is to double-click the relevant object type on the ORM diagram to bring up its Database Properties Sheet, select the Data Type category, and then hit Edit button to enter the new data type. For instance, if we select the CountryName object type in Figure 4, its properties dialog will appear as shown in Figure 7. Here I've set radio button to show the *physical data type* for the chosen DBMS, in this case SQL Server. You can change to a different DBMS by going to the main menu and choosing Database > Options > Drivers... and then selecting from the Database Drivers dialog box.

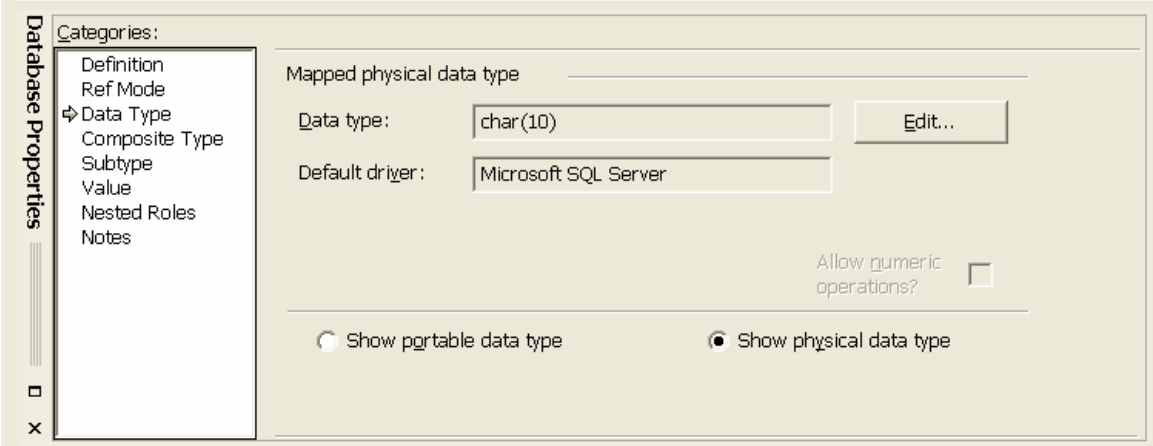

Figure 7 The data type can be set using the Database Properties window

The Data type field in the Database Properties sheet is read-only, so you cannot edit it directly. Instead, hit the Edit button to invoke the Data Type dialog box specific to that DBMS. The Native type and Length fields display the defaults (char and 10). To store country names as variable length character strings, select the relevant item (e.g. varchar) from the Native type drop-down list (scroll down, or quickly type the initial characters until the item appears), as shown in Figure 8(a). Click the mouse to accept the change, then move the cursor to the Length field. To allow for country names of up to 40 characters, change the value from 20 to 40 (see Figure 8(b)), then hit the OK button to accept the change. The Database Properties window now shows the Data type as varchar(40).

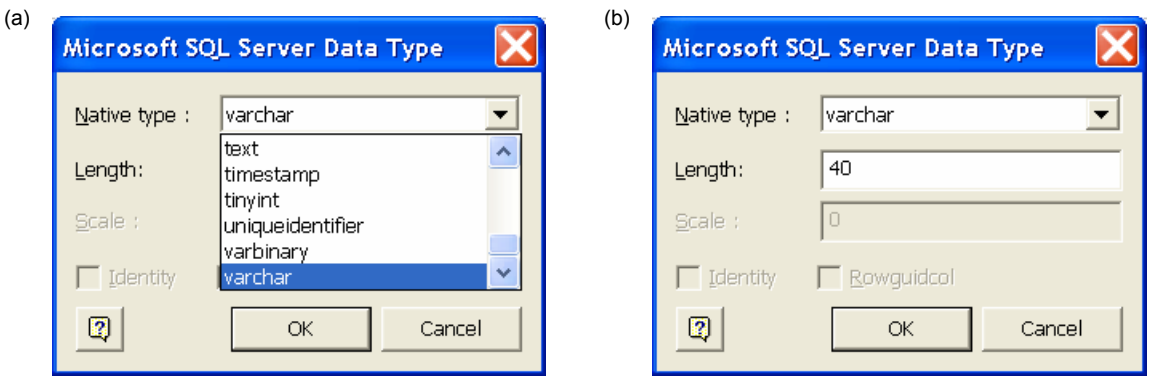

Figure 8 Changing the physical data type to varchar(40)

If you want to edit the data types for several object types, it's much quicker to do this via the Object Type pane of the Business Rules Window. If needed, open this window by selecting Database > View > Business Rules from the main menu. Choose the Object Types tab at the bottom to display the Object Types pane. The data type field supports a drop-down list for selecting the native type, and allows you to edit the length directly (see Figure 9). In fact, the whole field is editable, so you can simply type in the desired data type (e.g. varchar(40)) without accessing the drop-down list at all. The field is parsed as you go, with basic Intellisense support (e.g. if you type "varc" this is automatically expanded to "varchar"). If you enter an illegal data type, the Data Type dialog box discussed earlier is invoked for you to complete the entry there. Even if your typing skills are only basic, you should find that this in-situ (in-place) editing facility lets you make data type changes much faster. Try it now, setting the data type for GivenName and Surname to varchar(30).

In the Object Types pane, value types are depicted as broken ellipses, with light-blue fill. Entity types appear as solid ellipses with dark-blue fill. If an entity type has reference mode, its data type is the data type of the value type that references it. For example, the reference scheme Country(code) is an abbreviation for the association Country is identified by CountryCode, so when you set the data type for Country you are actually setting the data type for CountryCode. One standard way to reference countries is by their 2-letter ISO codes (e.g. 'AU' for Australia, and 'US' for the United States). This choice requires a fixed length character string of 2 characters in length, so we should edit the data type for Country to become char(2). In the US, social security numbers are eleven characters in length, requiring a data type of char(11). Value types often have additional constraints that are not expressible using built-in physical data types. For example, the characters in country codes must be letters, and social security numbers must match the pattern ddd-dd-dddd (where "d" denotes a digit). Currently, the modeling tool does not support the specification such pattern constraints, but they are trivial to implement in most DBMSs.

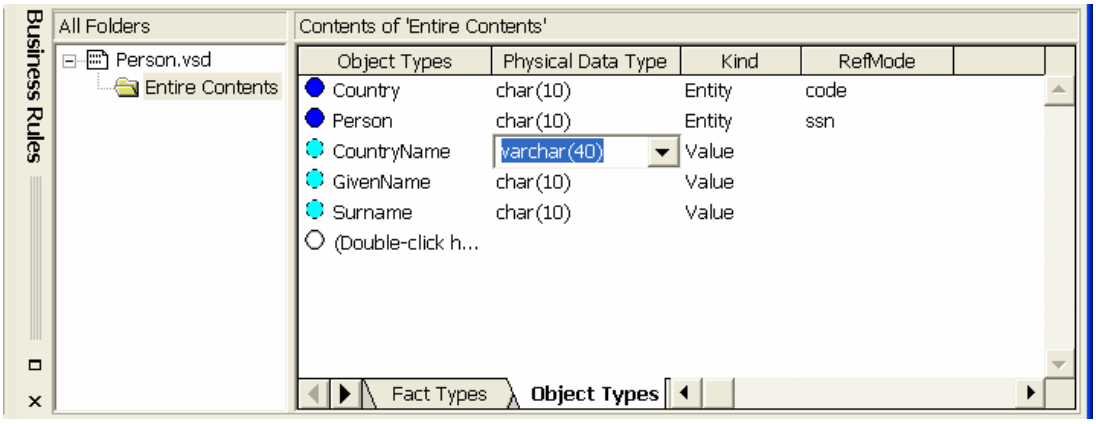

Figure 9 Data types can be edited quickly using the Object Types pane of the Business Rules window

If you already know the DBMS on which your model will be implemented, you will probably prefer to work with physical data types. However, if you haven't made that choice yet, or you intend to work with many kinds of DBMS then you may prefer to use *portable data types*. When portable data types are mapped to a given DBMS they are replaced by a corresponding physical data type. To display portable data types on the Data Type pane of the Database Properties window, select the "Show portable data type" radio button. Use the drop-down lists to select the appropriate values. For example, Figure 10 shows portable settings that might be used in place of the physical data type varchar(40). The Text category is used for character strings, the Type is set to Variable Length, the Size is set to Single Byte characters (rather than Double Byte), and the Length is set to 40. The possible settings for portable data types are summarized in Table 1. You can access details about specific settings by using the on-line help.

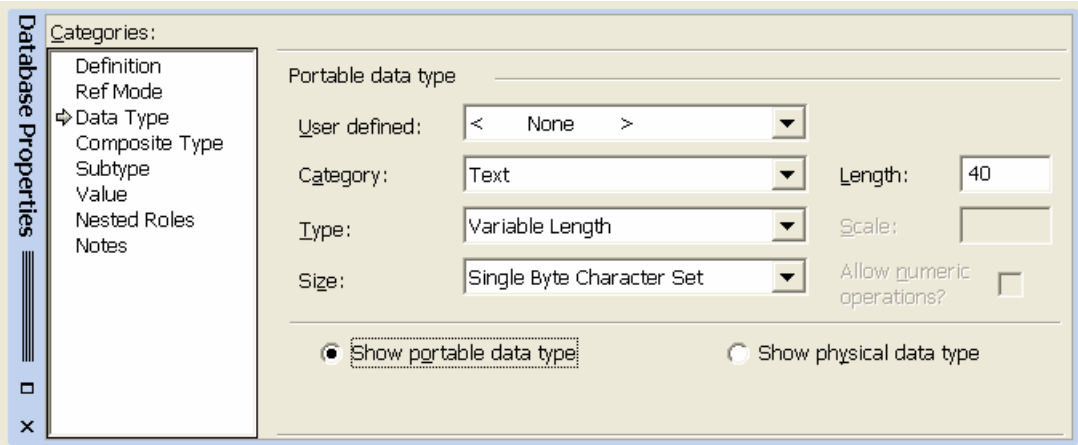

Figure 10 Portable data types may be used instead of physical data types

Table 1 Portable Data types

Category	*Type*	*Length/Precision (= default)*	*Scale (= default)*	*Size*	*Allow numeric op?*
User-defined	*typename*				
Text	Fixed length	Length = 10	--	Single byte char set	--
	Variable length	Length = 10	--	Double byte char set	--
	Large length	--	--		--
Numeric	Signed integer	--	--	Small, Large	Yes/No
	Unsigned integer	--	--	Small, Large	Yes/No
	Auto counter	Precision = 10	--	Small, Large	Yes/No
	Floating point	--	--	Small, Large	Yes/No
	Decimal	Precision = 10	= 2	Small, Large	Yes/No
	Money	Precision = 10	= 2	Small, Large	Yes/No
Raw data	Fixed length	Length = 10	--	--	--
	Variable length	Length = 10	--	--	--
	Large length	--	--	--	--
	Picture	--	--	--	--
	OLE object	--	--	--	--
Temporal	Auto timestamp	--	--	Small, Large	--
	Time	--	--	Small, Large	--
	Date	--	--	Small, Large	--
	Date & Time	--	--	Small, Large	--
Logical	True or False	--	--	Small, Large	--
	Yes or No	--	--	Small, Large	--
Other	RowID	--	--	--	--
	ObjectID	--	--	--	--
	Unknown	--	--	--	--

Conclusion

This article provided a brief overview of index constraints, constraint layers and data types. The next article will dig a little deeper into data types, and also discuss ways to control the generation of column names in the resulting relational model. If you have any constructive feedback on this article, please e-mail me at: thalpin@attbi.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.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.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*.
8. Halpin, T.A. 2002c, 'Microsoft's new database modeling tool: Part 6' (online at www.orm.net). This is a revised version of an earlier article of the same title in the May 2002 issue of *Journal of Conceptual Modeling*.
9. Rankins, R. et al. 2002, *Microsoft SQL Server 2000 Unleashed*, Sams Publishing.

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.