Version 4.0

Evaluation Tutorial

MetaCase Document No. ET-4.0

First Printing, 3rd Edition, January 2005.

MetaCase
Ylistönmäentie 31
FIN–40500 Jyväskylä
Finland

Tel: +358 14 4451 400
Fax: +358 14 4451 405
E-mail: info@metacase.com
WWW: http://www.metacase.com

You may order additional copies of this manual by contacting MetaCase or your sales representative.

Preface

The goal of this tutorial is to familiarize the reader with the features offered by MetaEdit+ Method Workbench. The reader will be provided with a step-by-step exercise on how to implement a modeling language and then how to use it. We will also take a look at how to extend the tool support by defining various documentation reports for our modeling language.

To complete this tutorial, the following are required:

- MetaEdit+ Method Workbench
- A web browser
- 1 hour of time to complete the basic exercise in Chapters 1 to 4 (Chapters 5 and 6 are optional and require more time to complete)

For further information about MetaEdit+, please refer to the 'MetaEdit+: User's Guide', 'MetaEdit Method Workbench: User's Guide' or our web pages at http://www.metacase.com.

1 Introduction to the Family Tree Modeling Language

To get a better picture of what MetaEdit+ is made for, let us walk you through the creation of a domain specific modeling language. Since the domains you're familiar with will be different for each evaluator, let's take one that everyone knows: the family. What kind of modeling language is good for modeling families? Of course we could try and (mis)use UML, but it's rather over-complicated for this, and we'd have to twist the semantics a fair bit. Instead, let's make our own, more like a traditional "family tree".

The first question to ask is what kinds of core concepts exist in our example Family Tree domain? As the Family Tree is a simple domain, we can quite soon figure out what we need. First there has to be a concept of *Person*. Each *Person* must have two other *Persons* as *Parents* and a *Person* can be a *Parent* to his or her *Children*. *Parents* and *Children* together form a *Family* relationship.

In addition to the basic concepts, we also need to know what information is stored with each concept. A *Person* for instance will need a *First name* and a *Last name*. Finally, we need to know what our diagrams will actually look like: we want to have symbols for our concepts. A quick sketch of a famous TV family might look Figure 1-1.

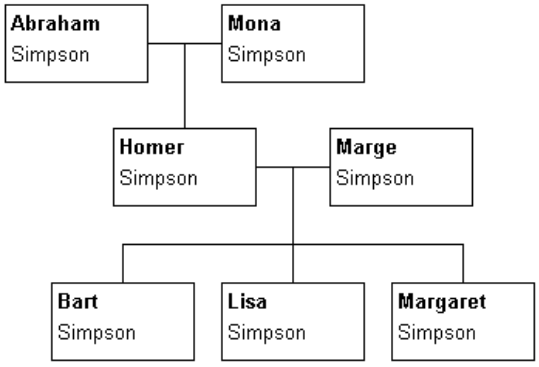

Figure 1-1. Sketch of a graphical notation for the Family Tree.

With the notation we now have all basic building blocks we need to start experimenting with our modeling language. What we need to do next is a proof of concept. So it is time to take a look at what these things look like in MetaEdit+.

2 Logging into MetaEdit+

MetaEdit+ is a repository-based application and therefore a connection with the repository must be established during the start-up. When started, MetaEdit+ opens a Startup Launcher as shown in Figure 2-1.

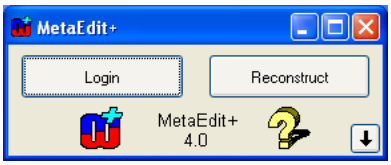

Figure 2-1. Startup Launcher.

Press the **Login** button in the Startup Launcher. If there is more than one repository available, MetaEdit+ shows a list of them (as in Figure 2-2). Choose the repository called 'demo' from this list and press **OK**. If 'demo' is not available, please install it from your MetaEdit+ distribution media.

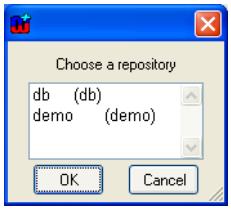

Figure 2-2. Choose a repository.

The next dialog asks for your user name and password. When MetaEdit+ is delivered, the default user name and password is 'user'. Fill these in and press **OK**.

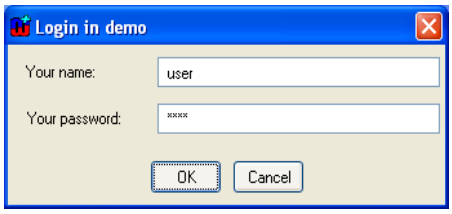

Figure 2-3. Entering user name and password.

After a successful login the Startup Launcher will close and you should see the main MetaEdit+ Launcher. If you are using an evaluation version of MetaEdit+ and this is your first login, a dialog will open for entering your evaluation license code. Enter the evaluation license code exactly as given in the evaluation instructions and press **OK**.

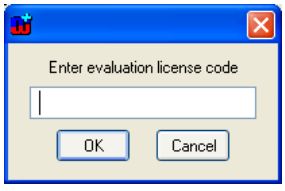

Figure 2-4. Evaluation license code dialog.

Together with the main Launcher you will get a dialog list where you can select the projects you want to open (Figure 2-5). Projects are the way data is organized in the repository. Some projects contain pre-defined methods (e.g. UML, SA/SD), others include models based on those methods (e.g. Tutorial, Examples). Choose just the 'Family Tree' project and click **OK**.

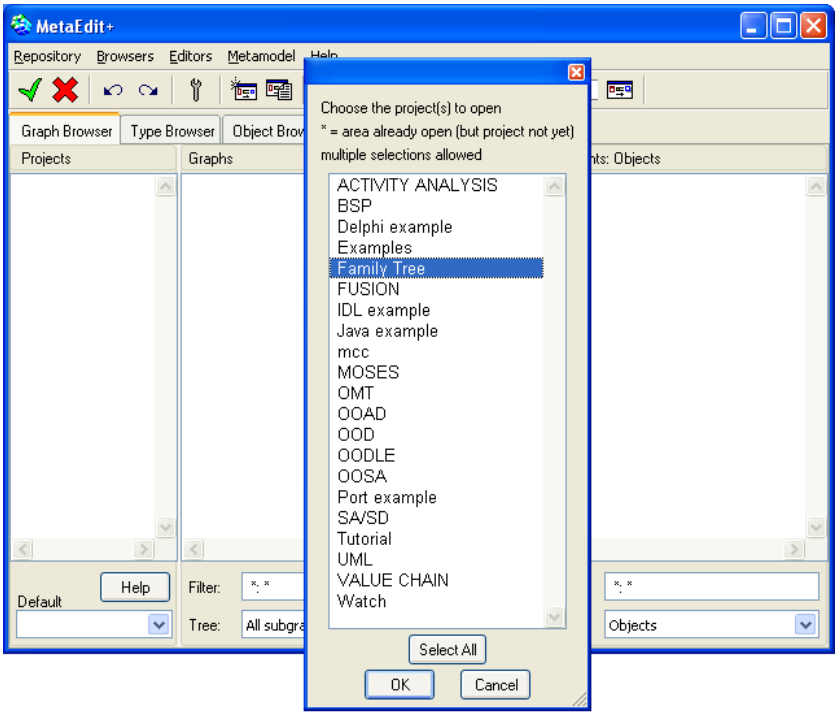

Figure 2-5. Choosing the project.

MetaEdit+ now asks for the project in which models created during this session should be saved. Choose 'Family Tree' and press **OK**. MetaEdit+ is now ready to be used. If you want to quit MetaEdit+ before completing this tutorial, please see Section 4.5 for exit instructions.

3 Implementing the Family Tree Modeling Language

The 'Family Tree' project we opened is empty: it does not contain any models or method definitions — yet. It is all clean and ready for us to implement our Family Tree modeling language.

This kind of activity, modeling a modeling language, is generally known as metamodeling. In MetaEdit+ we use a metamodeling language called GOPPRR. GOPPRR stands for Graph, Object, Property, Port, Relationship and Role, which are the metatypes we use when describing modeling languages. MetaEdit+ has a tool for creating and managing each metatype. You are not expected to be familiar with GOPPRR or its metatypes as these things are explained as they are needed as we complete this walk-through.

3.1 CREATING OBJECTS

The most obvious of the GOPPRR metatypes is **object**. The main elements of a diagram in your modeling language will generally be ***objects***. In our case, our domain concept of *Person* falls into this category. Start the Object Tool (as shown in Figure 3-1) by selecting **Metamodel | Object Tool** from the MetaEdit+ launcher.

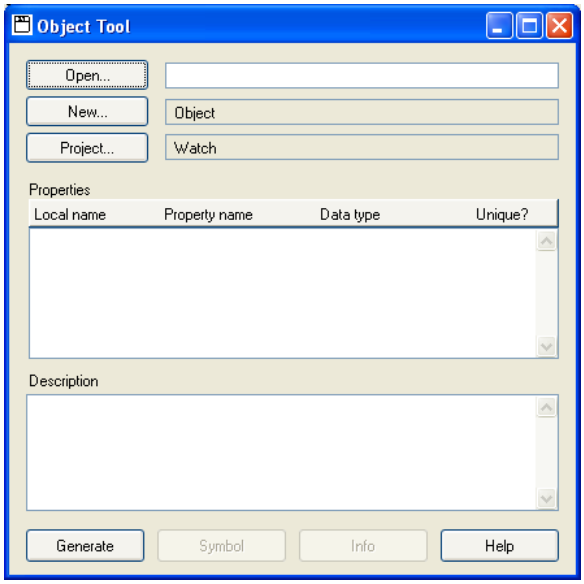

Figure 3-1. Object Tool.

Start creating the *Person* type by typing its name (i.e. 'Person') into the text field next to the **Open** button at the top of the Object Tool. Also make sure that the project for your object is 'Family Tree' (if not, you can change it by pressing the **Project** button and choosing 'Family Tree' from the list that appears).

Next, we need to state what information can be stored for each *Person*. In GOPPRR each slot for storing information is called a ***property***. To add a property to our Person object, move the mouse over the area of the **Properties** list in the Object Tool and press the right mouse button. You should now see a pop-up menu with two elements. Select **Add Property…** to add a property. As there are no property types defined yet, MetaEdit+ automatically chooses to create a new property type (if there are already some types, you must choose 'New Property Type' from the dialog that opens). This opens a Property Tool for defining a new property type (shown in Figure 3-2).

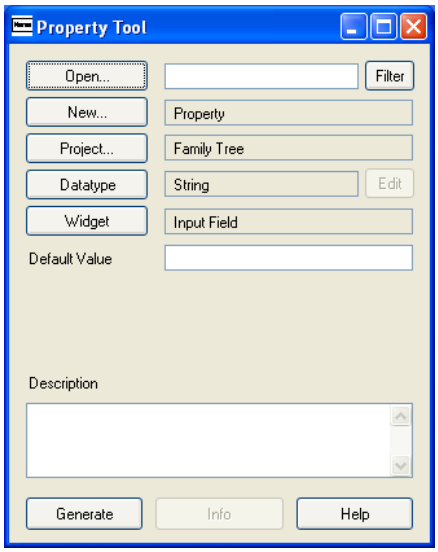

Figure 3-2. Property Tool.

In the Property Tool, enter the name for the property ('First name'). The other values are correct by default in this case, so you can proceed by pressing the **Generate** button. If you see a dialog asking "Do you really want to add the property First name to Person?', press **Yes** to accept the addition and accept 'First name' as local name in the following dialog. Close the Property Tool and go back to the Object Tool. You see now how the created property appears in the **Properties** list of the *Person* object (as shown in Figure 3-3).

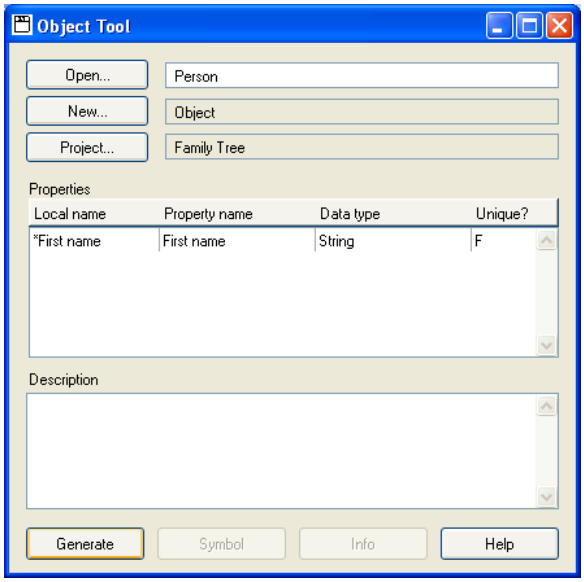

Figure 3-3. Object Tool with the 'First name' property.

Next we will create the 'Family name' property. In the Object Tool, again move the mouse over the **Properties** list and press the right mouse button to open the pop-up menu. Choose **Add Property…** from there, and select 'New Property Type'. In the Property Tool, enter 'Family name' in the Name field.

The **Widget** button in the Property Tool allows us to select how users will enter values of this property. As there will be several *Persons* with the same Family Name, it would be useful if the user could select these previously entered values from a pull-down list, into which each new name will be added. The widget for this is called an Editable List: press the **Widget** button and choose 'Editable List' from the possible widget types. Press **Generate** and close the Property Tool.

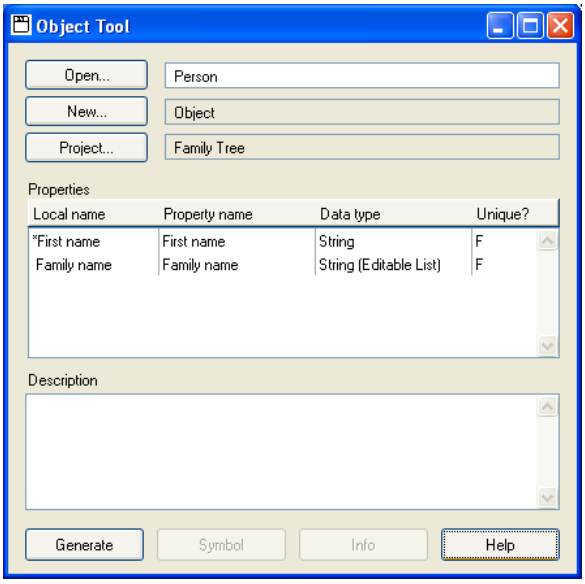

Figure 3-4. Object Tool with properties.

Your Object Tool should now look like Figure 3-4. Press **Generate** (answer 'Yes' if MetaEdit+ asks if a default dialog for the new object should be generated). You have now completed the definition of the *Person* concept.

3.2 CREATING SYMBOLS

As *Person* is a concept we are going to use in our graphical representations of family trees, we need to define a graphical symbol for it. You can start the Symbol Editor by pressing the **Symbol** button in the Object Tool. As creating good looking symbols from scratch may take a while, we skip some dirty work here this time and load the symbols from a symbol library.

In the Symbol Editor, select **Symbol | Browse Library…**, choose the symbol called 'Person' from the list and press **Paste & Close**. This inserts the predefined symbol template for *Person* into the editor (as shown in Figure 3-5).

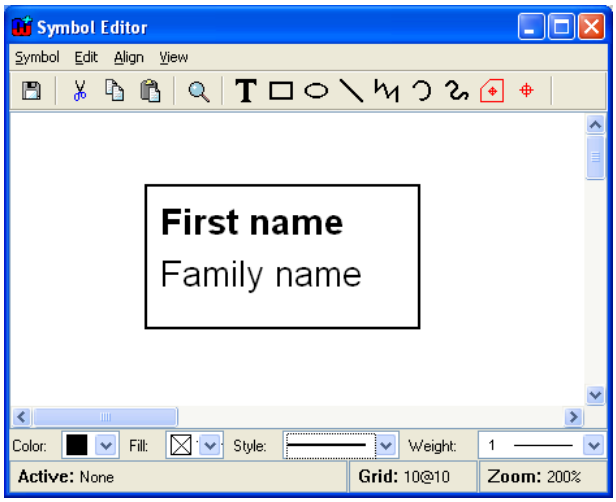

Figure 3-5. Symbol Editor with predefined *Person* symbol loaded.

We must, however, link the 'First name' and 'Family name' text fields to the corresponding properties we added to our *Person* object. Click the text 'First name' and press the right mouse button. The text field gets selected and a pop-up menu will be shown. Select **Format…** from the pop-up menu as in Figure 3-6.

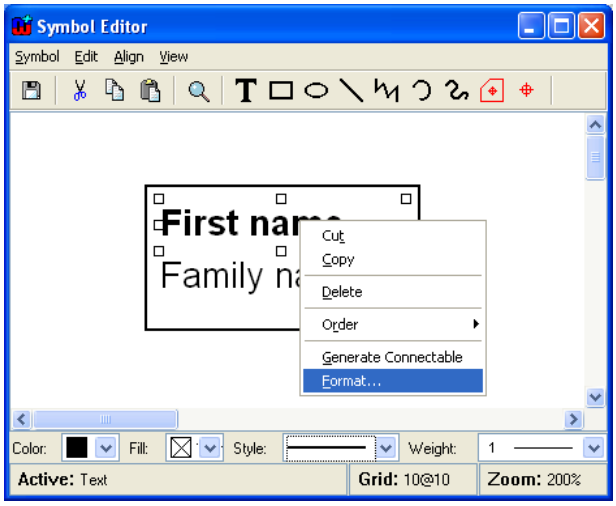

Figure 3-6. Formatting a text field.

A format dialog for the text field will open. Currently, the text field is showing the fixed text that is entered into the text box in the middle of the dialog. However, what we want to do is to make the text field show the value of the 'First name' property instead. To do this, click the **Property** radio button and choose 'First name' from the pull-down list next to it, as in Figure 3-7. Accept these changes and close the format dialog by pressing **OK**. The name of the selected property type now appears in the text field. In a similar way, set the content of the lower text field to be 'Family name'.

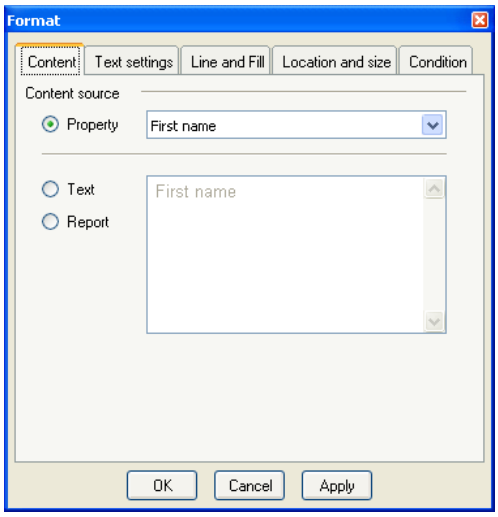

Figure 3-7. Format dialog for a text field.

After assigning the properties to the text fields, there is still one more thing to do. We must define the connectable area for our symbol, i.e. a perimeter the incoming connection lines will stop at. In this case the default connectable is suitable for our purposes, so you can let MetaEdit+ generate it for you by selecting **Symbol | Save** or by pressing the **Save** button in the Symbol Editor toolbar. A connectable will appear around the symbol elements as in Figure 3-8.

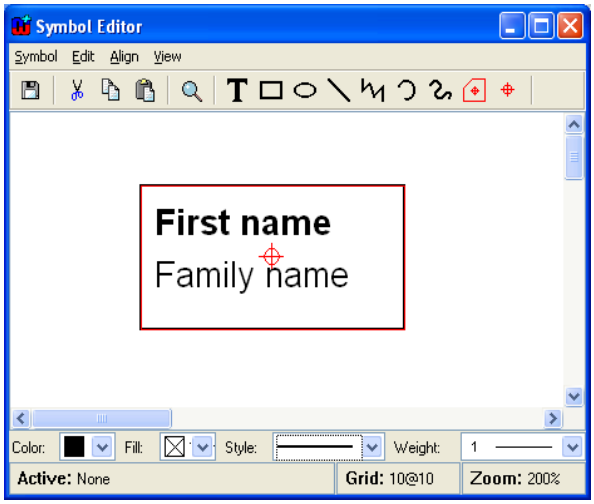

Figure 3-8. Symbol Editor with complete *Person* symbol definition.

The symbol is now ready and saved, so you can close the Symbol Editor by selecting **Symbol | Exit** or closing the editor window.

3.3 SAVING YOUR WORK

As we have now defined a significant part of our domain concepts, it is a good time to save our work. Go back to the MetaEdit+ launcher. There you can find **Commit** and **Abandon** buttons (as shown in Figure 3-9).

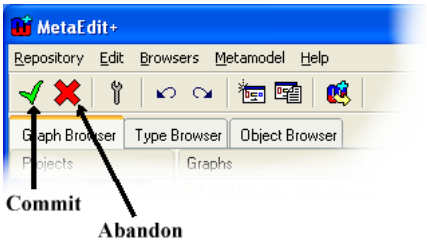

Figure 3-9. Commit and Abandon buttons in the MetaEdit+ launcher.

Pressing the **Commit** button saves the changes made into the repository, while pressing **Abandon** restores the state saved by the previous commit. Press **Commit** and wait for the operation to complete.

3.4 CREATING RELATIONSHIPS

What do we need to do next? Let's take a look at our problem domain specification again. We have now created the concept of *Person*. According to our sketch in Figure 3-10, we now have to define *Family*, a relationship between *Parents* and *Children*.

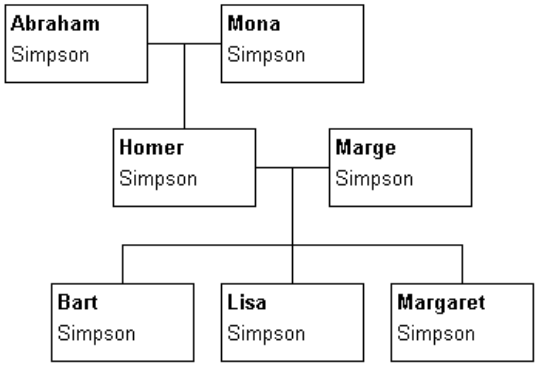

Figure 3-10. Sketch of a graphical notation for the Family Tree

In addition to the *Family* relationship, we need concepts to represent the ***roles*** a *Person* can play in that relationship. Simply, a *Person* can be in either a *Parent* role or a *Child* role in a given *Family*. Visually, each of the lines going towards a *Person* is a role, and the central meeting point of the lines is the relationship.

Looking at things a little more carefully, we can see there are certain rules on how we are allowed to bind together *Persons* in a *Family*. For instance, in any given *Family* relationship there must always be exactly two objects in *Parent* roles, but there may be any number of objects in *Child* roles.

To handle these kinds of issues, we need a new metamodeling structure known as a ***binding***. A binding contains the information on which objects can take part in a relationship in which roles, and how many times each role may occur. Each binding consists of one relationship, two or more roles and for each role, one or more objects.

First though we need to define the relationship and roles. Start the Relationship Tool by selecting **Metamodel | Relationship Tool** from the MetaEdit+ launcher. The Relationship Tool looks and works just like the Object Tool. To create a new *Family* relationship type, enter 'Family' in the top-most text field (as in Figure 3-11).

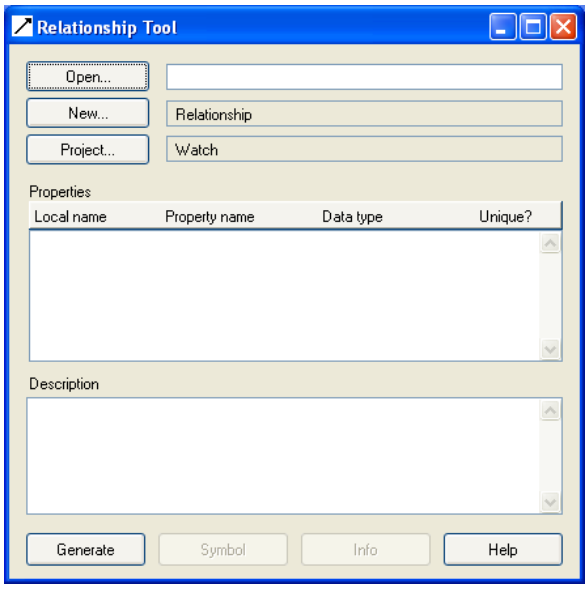

Figure 3-11. Relationship Tool

As we do not want to include any properties in our *Family* relationship at this time, you can complete the generation of the new relationship type by pressing **Generate**. As the default symbol provided by MetaEdit+ for this new type is sufficient for our purposes, we do not need to worry about the symbol here and we can close the Relationship Tool.

3.5 CREATING ROLES

Next we must define the roles we need. Start the Role Tool by selecting **Metamodel | Role Tool** from the MetaEdit+ launcher. The roles for our *Family* relationship are easy to create, as they do not require any properties or symbols. So, to create a *Parent* role, just enter 'Parent' in the top-most field (Figure 3-12) and press **Generate**.

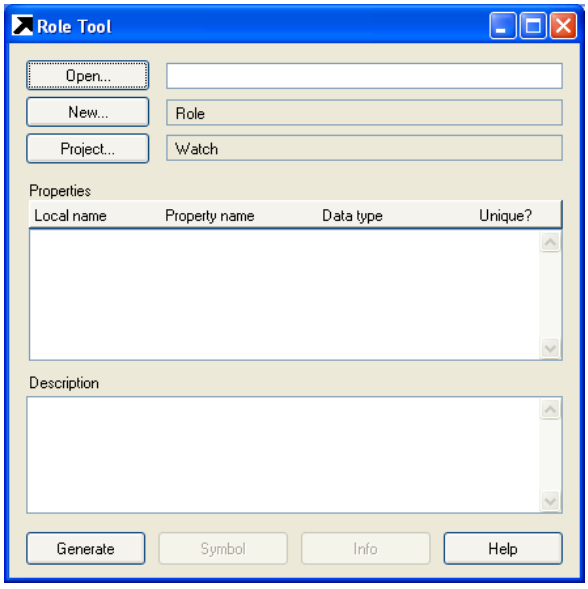

Figure 3-12. Role Tool.

The *Child* role is created in the same way. Close the *Parent* Role Tool (otherwise we would be making changes to the *Parent* role, rather than creating a new one). Start a new Role Tool, enter 'Child' in the top-most field and press **Generate**.

3.6 CREATING GRAPHS

We now have all the metamodel fragments we need for the first version of our Family Tree modeling language, so it could be a good time to commit again. What we need to do now is to somehow assemble these fragments into a complete modeling language. However, let us first consider again what we are about to do. What is our Family Tree modeling language actually like? It is a simple diagram notation for drawing family trees. This means that we should now be able to define the diagram type of the Family Tree modeling language.

In GOPPRR, there is a metatype ***graph*** that corresponds to a diagram type. To create a new diagram type, we must create a new graph type with the Graph Tool. Open the Graph Tool by selecting **Metamodel | Graph Tool** from the MetaEdit+ launcher.

First, enter the name for our modeling language ('Family Tree') in the **Name** field. Go to the **Properties** list, press the right mouse button and select **Add Property…** from the pop-up menu. This time we do not want to create a completely new property type, instead we shall reuse the existing 'Family name' property as a name for our diagram: the name of each diagram can thus be a family name. So, choose 'Family name' from the list of possible properties (if asked, accept the default local name proposed by MetaEdit+). Your Graph Tool should now look similar to Figure 3-13.

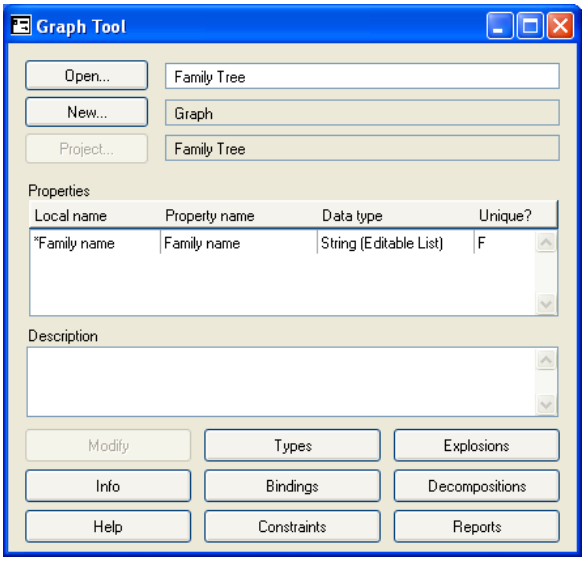

Figure 3-13. Graph Tool.

Press the **Types** button to open the Graph types definer. This is where you choose the types you want to use in your modeling language. Go to the **Relationships** list, press the right mouse button and select **Add…** from the pop-up menu. From the list that opens, choose 'Family' and press **OK**. The 'Family' relationship now appears in the **Relationships** list. Similarly, add 'Parent' and 'Child' roles to the **Roles** list (multiple selection is allowed) and 'Person' to the **Objects** list. Your Graph types definer should now look like Figure 3-14.

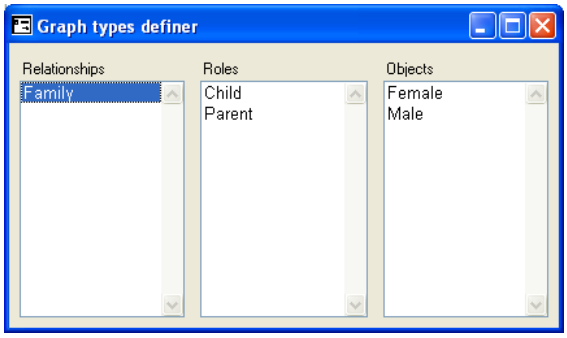

Figure 3-14. Graph types definer.

Close the Graph types definer and go back to the Graph Tool. Press the **Bindings** button to open the Graph bindings definer. Here you will create the bindings that actually connect your metamodel fragments as a complete modeling language specification. Go first to the **Relationships** list, press the right mouse button, select **Add…** from the pop-up menu and then 'Family' from the list.

Make sure that 'Family' is selected in the **Relationships** list. Go to the **Roles** list and press the right mouse button there. Choose **Add…** and then select 'Parent' from the list. Now, with both 'Family' and 'Parent' selected, add 'Person' to the **Objects** list (skip the **Ports** list). You have now created a part of the binding that says: "The *Parent* role in a *Family* relationship must connect to a *Person* object".

As there are always two *Parents* for a *Person*, add another *Parent* role similarly and connect it to the *Person* object. The Graph bindings definer should now look like Figure 3-15.

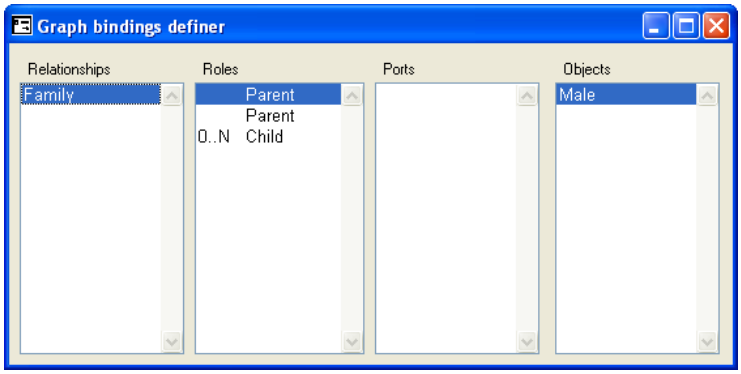

Figure 3-15. Graph bindings definer.

In addition to a *Parent* role, there is also a *Child* role in the *Family* relationship. To create it, move the mouse over the **Roles** list, press the right mouse button, choose **Add…** from the pop-up menu and then 'Child' from the list of available roles. As the **Objects** list now becomes blank, go there and add the *Person* object again. We have now created the part of a binding that says: "The *Child* role in a *Family* relationship must connect to a *Person* object".

As the problem domain specification implies, there are certain constraints that limit the set of possible *Family* combinations: there can be two and only two *Parents* but zero to many *Children* in a *Family*. In GOPPRR, these constraints are handled by roles. The requirement for two *Parents* is already taken care of with two separate *Parent* roles, but the cardinality of 'zero to many *Children*' must still be set. Select the *Child* role from the **Roles** list, press the right mouse button and choose **Cardinality…** from the pop-up menu. In the Cardinality Dialog that appears (Figure 3-16), set the **Minimum** to 0 and the **Maximum** to N and press **OK**.

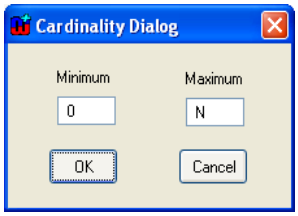

Figure 3-16. Setting binding cardinalities

Your Graph bindings definer should now look like Figure 3-17. Close the Graph bindings definer, press **Generate** in the Graph Tool (if asked, accept the generation of the default dialog). Also remember to commit your work again.

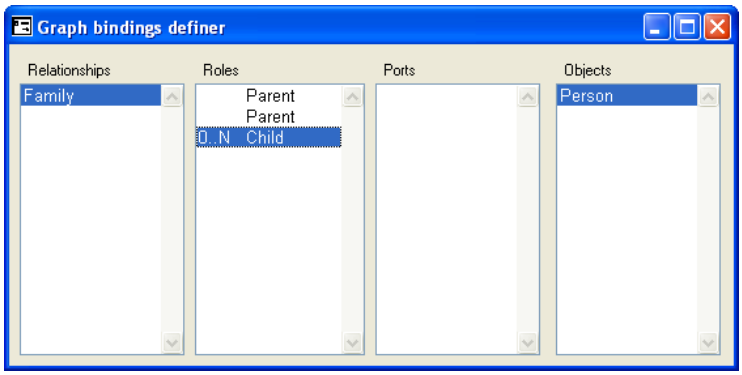

Figure 3-17. Graph bindings definer with more definitions

Congratulations! You have now completed the first version of our Family Tree modeling language.

4 Working with the Family Tree Modeling Language

So far we have been exploring the metamodeling tools of MetaEdit+. In addition to these, MetaEdit+ also offers a fully functional CASE environment for using the modeling languages we have created. Let us now familiarize ourselves with the CASE tool functionality by trying out the Family Tree modeling language.

4.1 CREATING A FAMILY TREE DIAGRAM

To start creating a new Family Tree diagram, select **Edit | Diagram Editor…** from the MetaEdit+ launcher. Usually MetaEdit+ asks you to choose the graph to open, but as there are no existing graphs available yet, MetaEdit+ automatically starts the creation of the only available graph type, 'Family Tree'. You are asked to enter the name of the family for which the Family Tree diagram will be created (in our example in Figure 4-1 we created the tree for the Simpson family as we know it from TV).

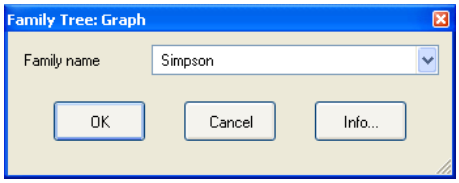

Figure 4-1. Entering the Family name for a new Family Tree

Press **OK** and MetaEdit+ opens an empty Diagram Editor for drawing a Family Tree (as in Figure 4-2). At the top of the Diagram Editor window you will find the usual menu bar. Below that is an action toolbar, and below that another toolbar that contains the types for the modeling language in use (the objects are on the left side and relationships on the right side). The main part of the window is reserved for the drawing area, and there is a status bar at the bottom that shows the name and type of the selected element.

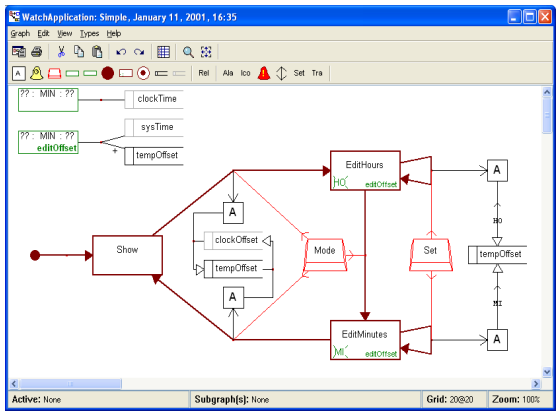

Figure 4-2. The Diagram Editor

4.2 ADDING OBJECTS TO THE FAMILY TREE DIAGRAM

In the Diagram Editor, press the leftmost button on the toolbar that contains object and relationship types. You have now selected the *Person* object type to be used. To place a *Person* object in our diagram, move the mouse over the drawing area and click the left mouse button. You can now enter the information about the object you are creating in the dialog that opens (as in Figure 4-3). Enter the first name and family name for the Person you are about to create (please note that you can also choose the family name from the list with the mouse) and press **OK**.

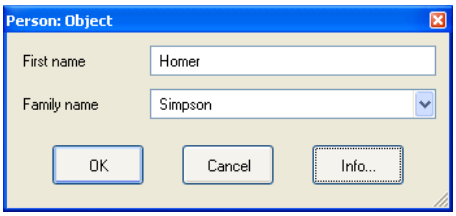

Figure 4-3. Entering information about an object

The object is now shown with its properties in the drawing area of the Diagram Editor. Now create a few more objects in the same way. Moving the symbols around and resizing them works the same way as in any drawing tool. To change the property values of a symbol double-click the symbol to bring up the property dialog. When you have drawn a few more objects, your Diagram Editor should look something like this:

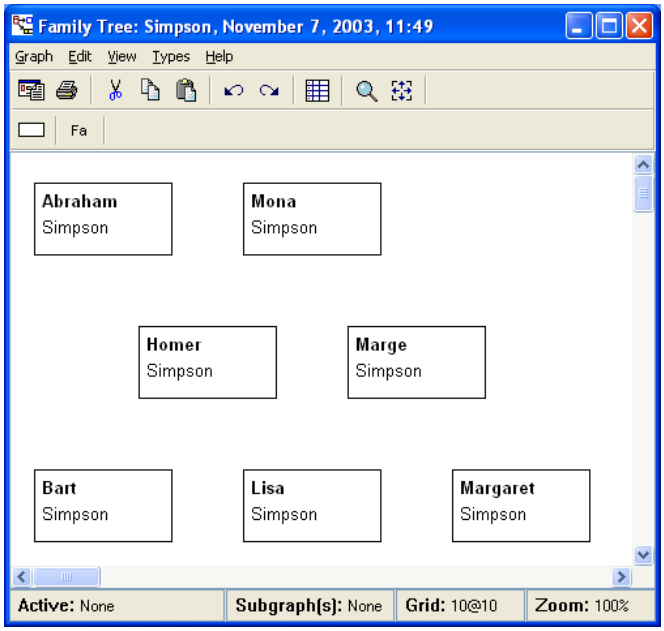

Figure 4-4. The Diagram Editor with some objects

4.3 ADDING RELATIONSHIPS TO THE FAMILY TREE DIAGRAM

The next things we want to add to our diagram are the relationships between parents and children. There are two ways to draw relationships. The first way is applicable when we want to draw relationships between three or more objects. The second way is a shortcut for when only two objects are involved. Let's look at the first way as we create the Family relationship between Abraham, Mona, and their son Homer.

First, press the button labeled 'Fa' in the relationship toolbar to trigger the creation of a 'Family' relationship. Move the cursor over the *Person* object that is about to be in the first *Parent* role so that the connectable is highlighted (step 1 in Figure 4-5) and click the left mouse button. Then click again in the space between the two parents, to set the midpoint of the relationship. (step 2). Next click the second parent (step 3), and then the child (step 4). Finalize the creation of the relationship by clicking the right mouse button (alternatively, the relationship can be finalized by double-clicking the left mouse button when connecting the last object). The relationship between the chosen objects now appears in the Diagram Editor.

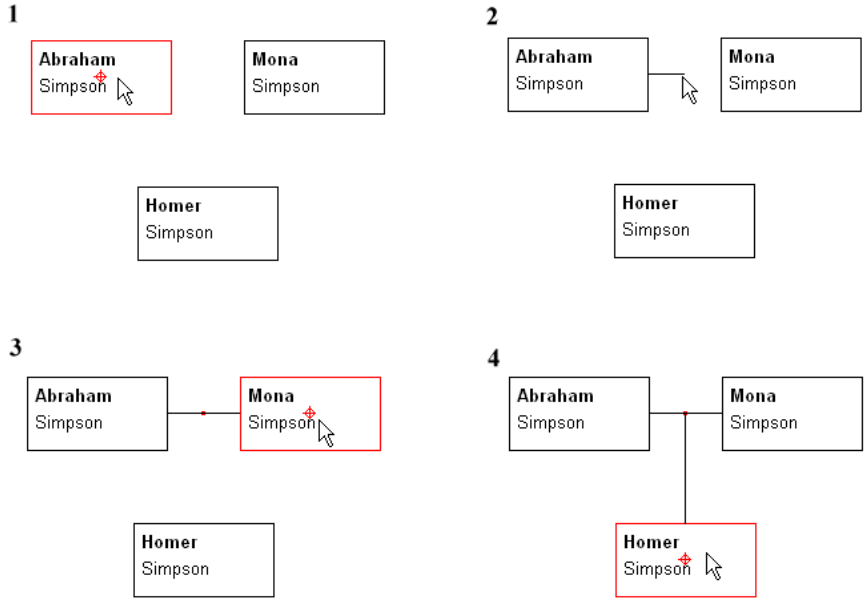

Figure 4-5. Creating 'Family' relationship.

Let's summarise that first way of creating a relationship. You click four times thus – first parent, empty space, second parent, child – and then right-click to finalize the relationship.

The second way can be used when we want to draw a relationship that only includes two objects. A good example here is a family without children; in our example, let's connect Homer and Marge. Again, start the creation of a 'Family' relationship by pressing the 'Fa' button in the relationship toolbar. Move the cursor over the first object for the relationship, Homer. Press and hold down the left mouse button, drag the cursor over the other object, Marge, and release the button. A binary relationship between the objects will now appear.

You can also add children to this relationship later. Take a look at the relationship you have just drawn. There is a small red dot in the middle of the line. Click to select that dot, and then press the right mouse button. Select **Add a New Role…** from the pop-up menu that opens.

Move the cursor over the object you want to connect in the *Child* role (e.g. Lisa) and click the left mouse button. You can also click in empty space along the way to the object, to define the path the role will take. For example to add a 'dogleg' to Bart you would select the relationship, choose **Add a New Role…,** then click above Lisa, above Bart, and finally on Bart.

You can now continue to complete the Family Tree diagram by adding a role to Margaret so that it resembles the diagram in Figure 4-6. Remember to commit your work!

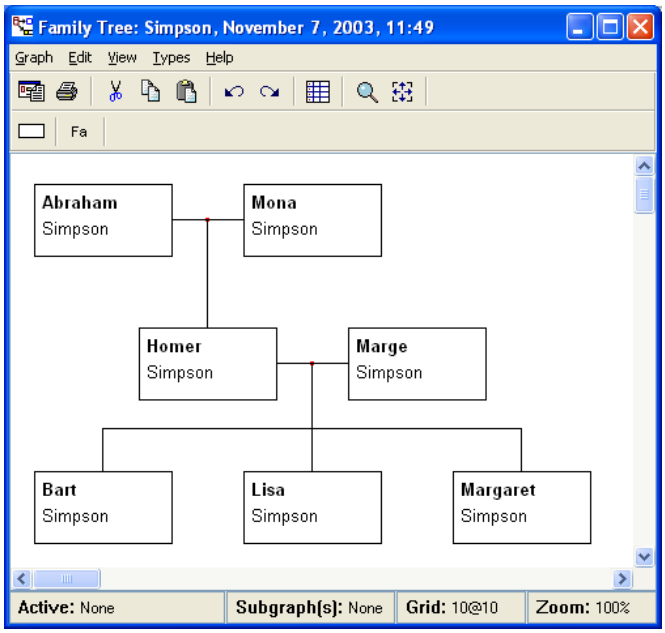

Figure 4-6. The finished Family Tree diagram

4.4 GENERATING REPORTS FROM THE FAMILY TREE DIAGRAM

We have now created a graphical language to model family trees and have been able to draw our first Family Tree diagram with it. Is there anything else we can do?

In many cases it would be very desirable to be able to produce various outputs of the information stored in a Family Tree diagram, e.g. as web pages, who's who entries etc. This can be done with MetaEdit+'s report generator. As an example, let's generate a web page from our Family Tree diagram.

Select **Graph | Reports | Run…** from the menu bar of the Diagram Editor containing our 'Simpson' Family Tree diagram (or press the **Run Report** button in the Diagram Editor's action toolbar). A list of the available pre-defined reports will appear. Choose 'Export graph to HTML' from the list (as in Figure 4-7) and press **OK**.

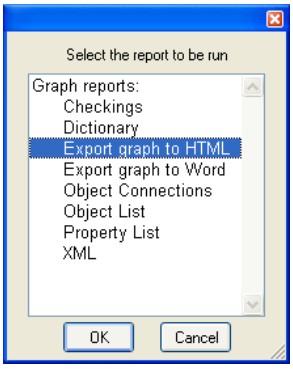

Figure 4-7. Default reports

MetaEdit+ now produces the HTML output, reports the success of the generation and opens the generated web page in your default browser (Figure 4-8).

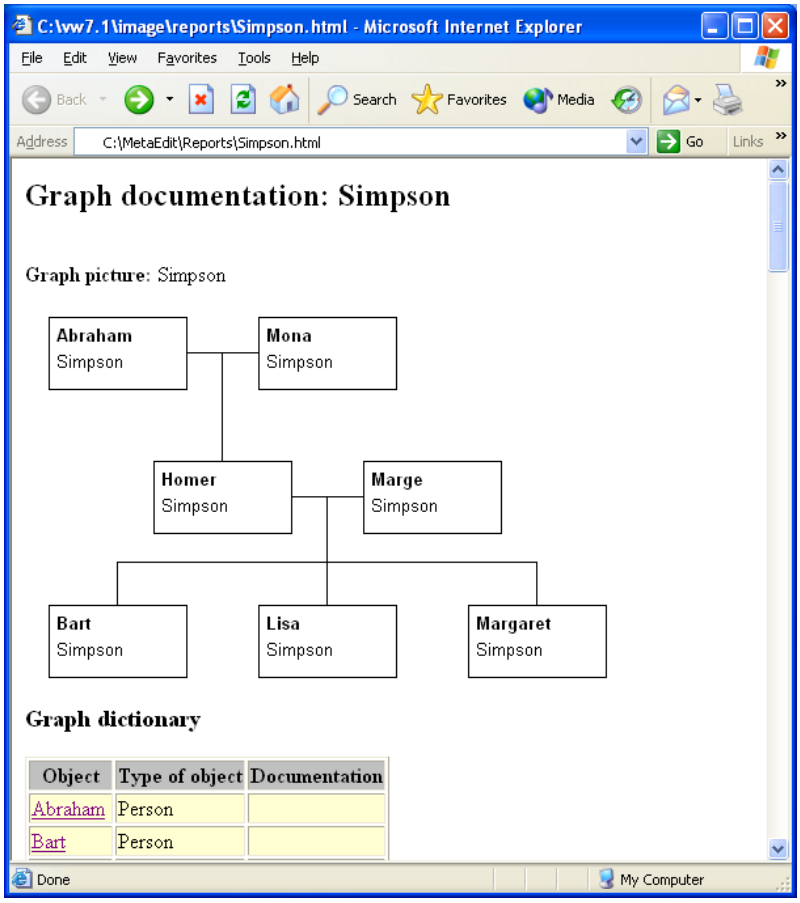

Figure 4-8. The results of the HTML generation

4.5 EXITING METAEDIT+

You have now completed the proof of concept cycle in developing the Family Tree modeling language. If you want to quit this tutorial at this time, you can exit MetaEdit+ by selecting **Repository | Exit…** from the main Launcher menu. MetaEdit+ will ask for confirmation before exiting. If you have uncommitted changes, MetaEdit will show the dialog in Figure

4-9. Choose **Commit and Exit** or **Abandon and Exit** depending on whether or not you want to save the latest changes.

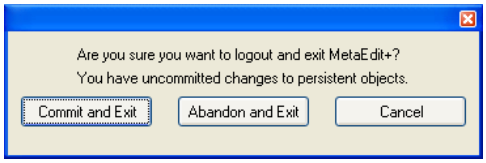

Figure 4-9. Logout with uncommitted changes.

However, if you are interested in seeing how the Family Tree modeling language can be developed further, start MetaEdit+ again and continue with the next chapter. If not, skip straight to the conclusion in Chapter 6.

5 Improving the Family Tree Modeling Language

We have now successfully designed, implemented and tested the first version of our Family Tree modeling language. This is, however, still only the beginning. Usually at this stage we have learned certain new things about our problem domain that require us to change, or at least to fine-tune, our modeling language. Let us now look at a couple of improvements for our Family Tree modeling language, and get familiar with some of the more advanced metamodeling and modeling concepts provided by MetaEdit+

5.1 PERSON AS MALE AND FEMALE

In our Family Tree language, we now have one concept of *Person* with properties of *First name* and *Family name*. We know, however, that in most cases we probably would like to store more information than that about each *Person*. For most applications it would be desirable to know dates for a *Person's* birth and death. Furthermore, as a female *Person* may change her family name when she gets married, storing the maiden name separately for a female *Person* would be a useful feature. In diagrams, we could also want to have a different symbol for male and female persons.

Now all of this obviously calls for a specialization of the concept of *Person* into concepts of *Male* and *Female*. However, as these two new concepts are straight subtypes of *Person*, we want to present this semantic fact in our metamodel also. Thus we assign all common properties to the concept of *Person*, define *Male* and *Female* as its subtypes and add additional properties to *Male* and *Female* where needed.

Start the Object Tool by selecting **Metamodel | Object Tool**. In the Object Tool press the **Open…** button and select 'Person' from the list dialog. You now have the definition for *Person* in your Object Tool. Go to the **Properties** list and add a property, choosing to create a new property type. In the Property tool, enter 'Date' as the name. As the default data type (string) and widget (input field) are correct, press Generate (accept the addition of the property if asked). Now, close the Property Tool and go back to the Object Tool. Select the newly created 'Date' property from the property list and open its pop-up menu by pressing the right mouse button. Select **Local name…**, enter 'Date of birth' into the dialog that opens and press **OK.**

The local name was used here as a mechanism for reuse: the same property may have several local names to make it more suitable for various purposes. This will be further demonstrated as we create the 'Date of death' property. As we have already defined a property of Date, we do not now need to create a completely new property type, but we can reuse the existing one. First, make sure no property is selected in the Property list (so the new property will be added at the end), and choose **Add Property…** from the Property list pop-up menu. This time choose property type 'Date' from the list instead of 'New Property Type', and change its local name to 'Date of death' the same way you did with 'Date of birth'. Now you have reused one

property type, ‘Date’, as both the date of birth and the date of death. Your Object Tool should now look similar to the one in Figure 5-1.

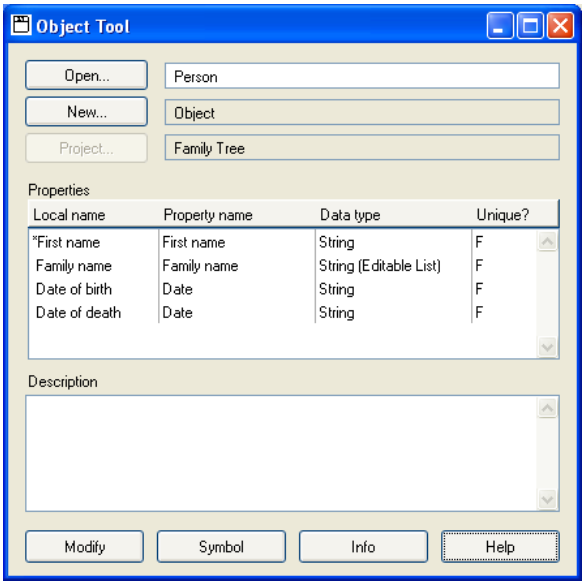

Figure 5-1. Object Tool with more properties.

Press the **Modify** button to accept the modifications. What we want to do next is to create new types *Male* and *Female* as subtypes of *Person*. In the Object Tool, click the **New…** button, and select ‘Person’ from the list of possible ancestor types. You are now creating a new Object type with properties inherited (and thus shown red) from the *Person* object. As the concept of *Male* actually adds nothing to the concept of *Person*, we can now create the *Male* object by entering ‘Male’ as the name in this new Object Tool. Then press **Generate** (if asked, accept the generation of the default dialog).

As *Male* is a concept which we are going to use in our diagrams, we need to define a graphical symbol for it. Start the Symbol Editor, load the symbol called ‘Male’ from the symbol library and modify it to use true properties instead of fixed text for the four text fields named after properties (leave ‘Birth’ and ‘Death’ as fixed text), as you did with the original ‘Person’ symbol. Save the symbol and close the Symbol Editor.

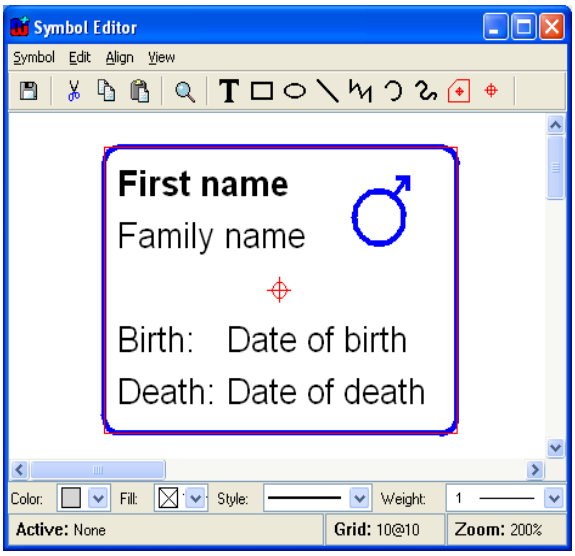

Figure 5-2. Symbol Editor with the completed symbol for *Male*.

Creating the *Female* object requires a little more work. Start again by creating a new type as a descendant of *Person*. Enter 'Female' as the name for this new descendant, and add an extra property (reuse the 'Family name' property type), then give it a local name of 'Maiden name'. At the end, the Object Tool for the *Female* object should look like Figure 5-3.

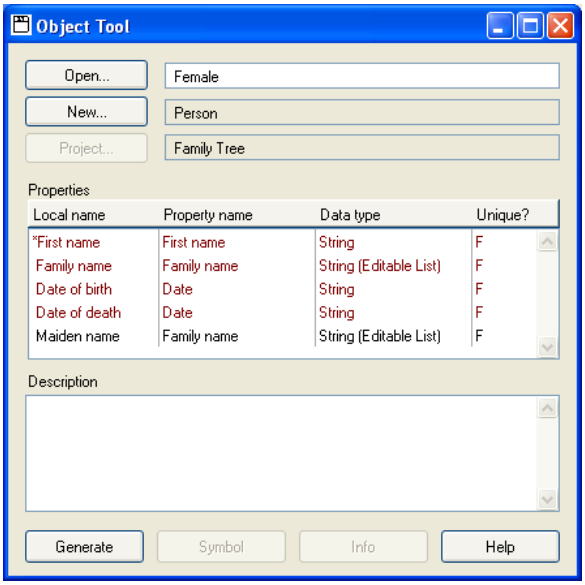

Figure 5-3. Object Tool for *Female* object

Press **Generate**, (allow the default dialog to be created if asked), and create the symbol for the *Female* object in the same way you did for *Male*: load the 'Female' symbol from the symbol library, assign real properties for the five appropriate text fields and save the symbol. You can now close the Symbol Editor and the Object Tool.

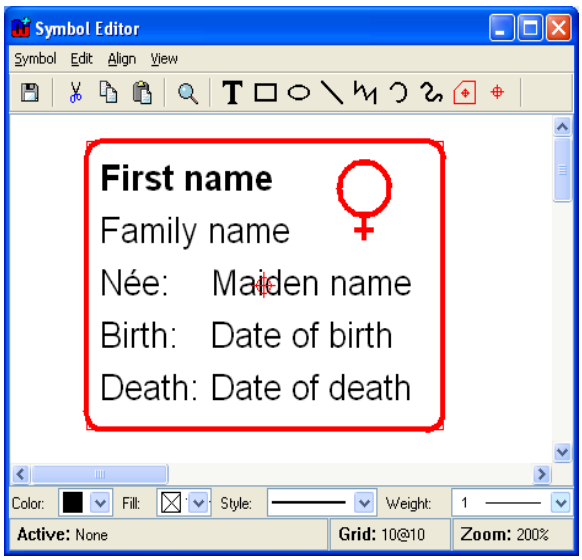

Figure 5-4. The Symbol Editor with symbol for *Female*

Commit your work and start the Graph Tool by selecting **Metamodel | Graph Tool** from the MetaEdit+ launcher. In the Graph Tool, press the **Open…** button. 'Family Tree' should be currently the only graph definition we have and therefore it should be loaded automatically. However, if there are more definitions, choose 'Family Tree' from the list that opens.

Press the **Types** button to open the Graph types definer. In the **Objects** list, select 'Person', press the right mouse button and select **Delete** from the pop-up menu. Then press the right mouse button again, select **Add…** and choose 'Male'. Add 'Female' similarly to get to the state in Figure 5-5 and close the Graph types definer.

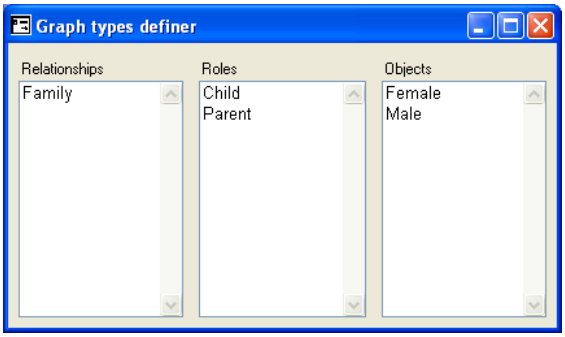

Figure 5-5. Modifications in Graph types definer.

Start the Graph bindings definer by pressing the **Bindings** button in the Graph Tool. Choose the first *Parent* role from the **Roles** list. Go to the **Objects** list, delete 'Person' and add 'Male' in its place. Similarly, select the second *Parent* role and replace its *Person* object with *Female*. Your Graph bindings definer should now look similar to Figure 5-6.

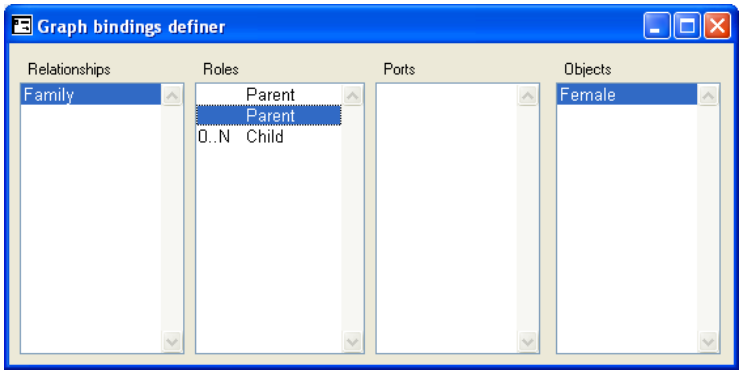

Figure 5-6. Modifications in Graphs bindings definer.

As for the *Child* role, leave it connected to the *Person* object. While *Person* was actually removed from the type set of our Family Tree modeling language, its references were not and thus we can use it here. *Person* as their super type allows both male and female to be used.

Close the Graph bindings definer and press **Modify** in the Graph Tool. The modifications for improving our modeling language have been now made, so you can try it out again. Since this change was fairly radical, you will have to delete all the old instances and create new ones: the old objects were all of type Person, and now you want them to be Male and Female instead.

5.2 ADDING REPRESENTATIONAL PRECISION WITH PORTS

Objects, relationships and roles are reasonably strong mechanisms for presenting the semantics between various model elements. However, sometimes even more precise semantics or behavioral rules are required for bindings. For this purpose, the GOPPRR metamodeling language defines a concept of **Port**. Port is a concept that can be thought of as

part of an object to which roles connect. Using ports we can provide additional information or constraints on how roles and objects are connected.

While ports are mostly intended for use on the conceptual level, they can also be utilized for advanced representational structures. For example, one visualization-related requirement for our Family Tree language could be that each *Child* role must be connected only to the top edge of the 'Person' symbol and each *Parent* role only to the side edge (right for Male, left for Female). This would constrain users to draw diagrams in the way we did in Figure 1-1, with the father on the left and mother on the right, a line between them as parents, and from the midpoint of that line descending lines to children. With our current binding definition such precision is not possible, so let us see what kinds of additions would make it possible.

What we need to do is to define two new connectables for our symbols and give them respective *Parent* and *Child* ports. First of all, though, we must define a new port type. Start the Port Tool by selecting **Metamodel | Port Tool**. Enter 'Family Port' in the top-most field and add a new property type called 'Port type' (with data type of String and widget of Input Field). The Port Tool should now look like in Figure 5-7. Accept the generation of the new port type by pressing **Generate**, and quit the Port Tool.

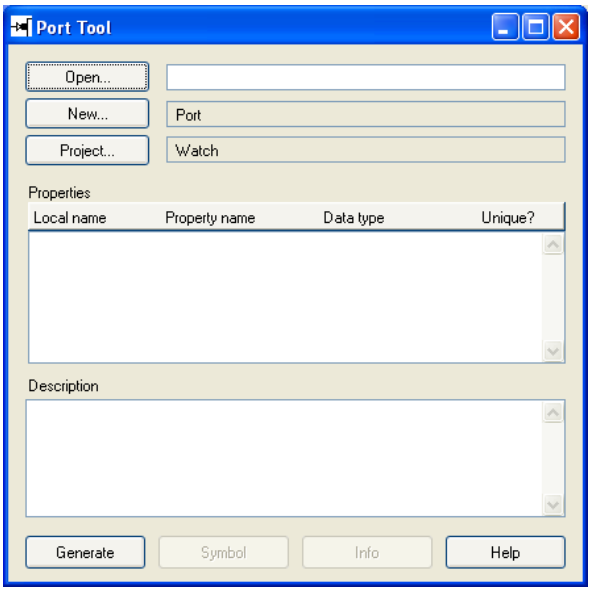

Figure 5-7. Port Tool.

Open an Object Tool then the Symbol Editor for the 'Male' object. To make the drawing of new connectables easier, remove the default one by selecting it (click the red cross-hair in the middle of the symbol) and choosing **Delete** from its pop-up menu. Activate the Connectable tool from the toolbar (the red polygon with a cross-hair in it on the right). Move the cursor over the top-left corner of the symbol, click the left mouse button, move the cursor to the top-right corner and double-click the left mouse button. A new connectable should now appear as in Figure 5-8 (blue parts of the symbol are shown as light-grey in the figure to emphasize the new connectable).

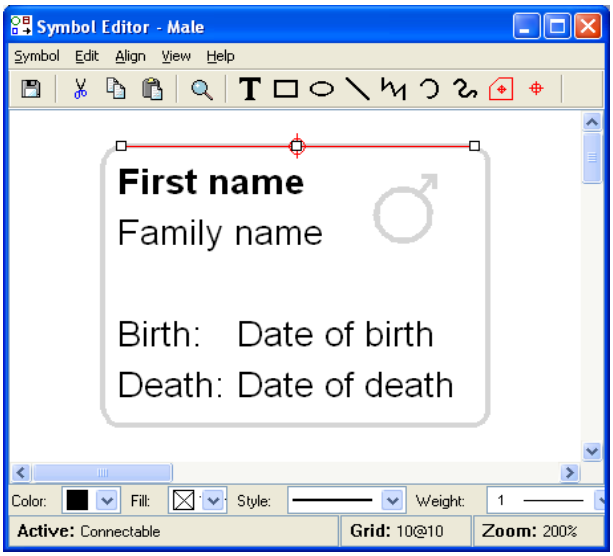

Figure 5-8. Creating a new connectable.

Keep the new connectable selected and select **Format…** from its pop-up menu. In the Port(s) list, press the right mouse button and select **Add…** from the pop-up menu that opens. Enter 'Child' in the **Port type** field and press **OK**. The *Child* port now appears in the **Port(s)** list as in Figure 5-9. Press **OK** to accept and close the format dialog.

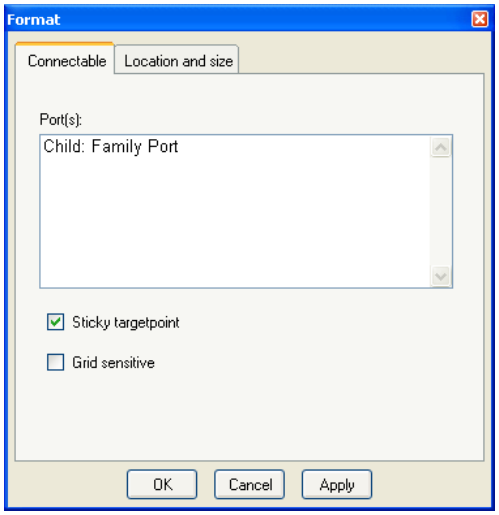

Figure 5-9. Attaching a port to the connectable.

Now create another connectable in a similar manner for the right edge of the symbol and give it a new Family port called *Parent*. The symbol definition should now look like Figure 5-10. Save the symbol and quit the Symbol Editor.

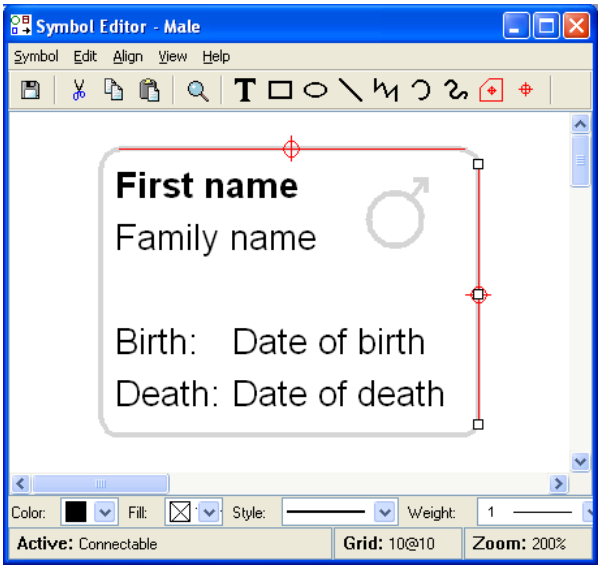

Figure 5-10. 'Male' symbol with two connectables.

Next, open the Symbol Editor for the 'Female' object. Create a connectable along the top edge as you did for the 'Male' symbol, but do not create a new port for it. Instead, add the existing *Child* port to it by selecting **Add Existing…** from the **Port(s)** list's pop-up menu in the connectable format dialog and choose the existing *Child* port from the list that opens. Next, make a new connectable down the side like for Male, but this time along the left edge of the symbol, and give it the existing *Parent* port. It is important to use the existing ports rather than creating new ones: otherwise you will not be able to create relationships between these objects. The symbol for the 'Female' object should now look like in Figure 5-11.

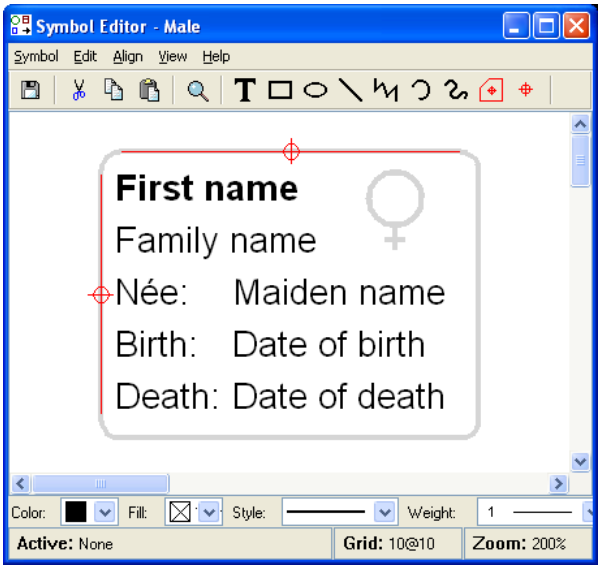

Figure 5-11. 'Female' symbol with two connectables.

Once you have made these additions to the symbols, open the Graph Tool for the 'Family Tree' graph type and open the Graph bindings definer. Select the *Parent* role from the **Roles** list, go to the **Ports** list, press the right mouse button and select **Add…** from the pop-up menu. In the list dialog that opens, double-click the 'Family Port' type and then select its instance, 'Parent: Family Port' from the next list dialog that appears. The Graph bindings definer should now appear as in Figure 5-12.

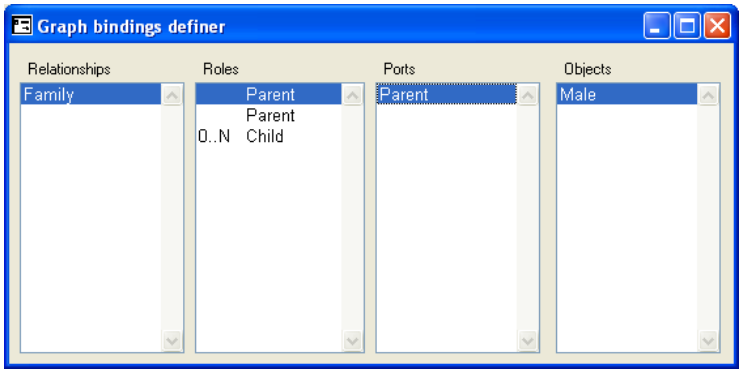

Figure 5-12. Graph bindings definer with port definition.

Similarly, attach the *Parent* port to the other *Parent* role, and the *Child* port to the *Child* role. You can then close the Graph bindings definer and accept the changes to the graph type by pressing **Modify** in the Graph Tool.

Now let's try out this latest version of the Family Tree modeling language. Delete the relationships you created before, and recreate them as in Figure 5-13. Note that when clicking an object while creating a relationship, that port is selected whose connectable's cross-hair is closest to the cursor. To make sure that the correct port is selected, you should thus click near to the edge you want: the right edge for a father, left for a mother, and top for a child. You will notice that unlike before, when role lines could move freely around the symbol while you moved related objects, all *Child* roles now remain connected to the top edge of the symbols. Similarly, *Parent* roles stick to the left or right edge of the parent symbols.

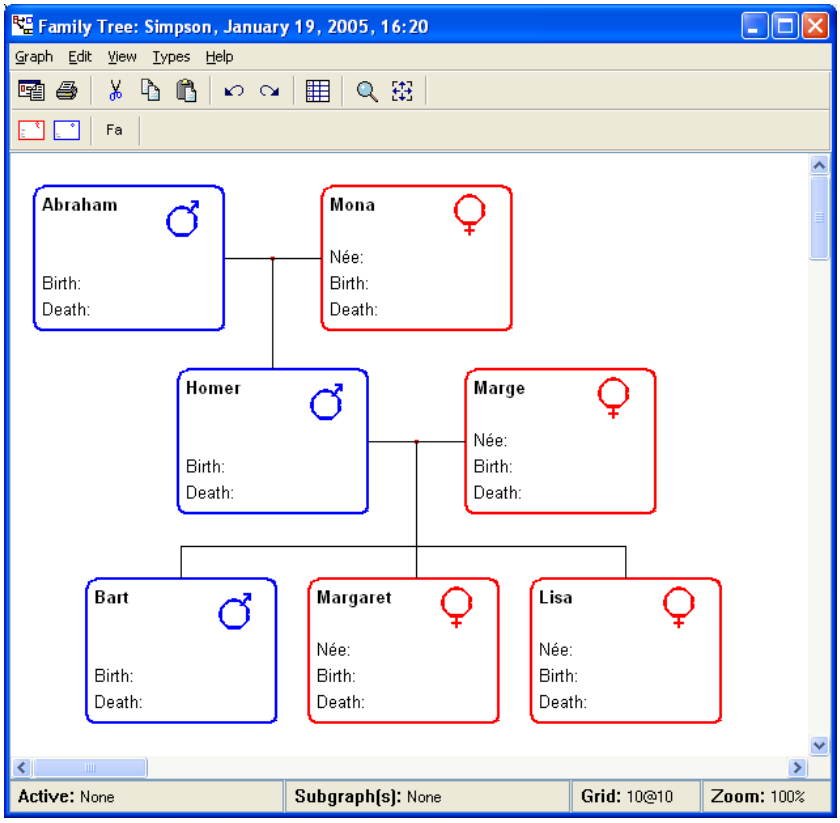

Figure 5-13. Family Tree with ports.

If you notice that although you are using the grid, role lines are no longer precisely horizontal or vertical, the probable cause is the position of the connectables' cross-hairs. These form the target points to which role lines point. If, say, the Child cross-hair is not directly above the symbol's center, as shown by the cross-hair of the default connectable in the middle of the symbol, role lines to the Child port will not be exactly vertical. You can correct the position of the cross-hairs in the Symbol Editor. Select the cross-hair at the centre of the symbol, and with shift down add the cross-hair for the Child connectable. Then choose **Align | Align Center** to shift the Child connectable along so it is centered over the default connectable's cross-hair. The Parent connectable can be corrected similarly, but using **Align | Align Middle** to shift it up or down.

5.3 CREATING REPORTS

Previously we generated predefined reports for our Family Tree diagram. However, to get the most out of our modeling language, we can make new domain-specific reports. Let us now explore how you can create reports of your own. Please note that if you have used different names for your metatypes than those presented in this tutorial (like 'Given name' instead of 'First name') while creating the metamodels, you have to use your own naming conventions in these example reports too!

In the Diagram Editor, select **Graph | Reports | Create** to open the Report Browser. The Report Browser, as shown in Figure 5-14, is a tool for creating, maintaining and running reports.

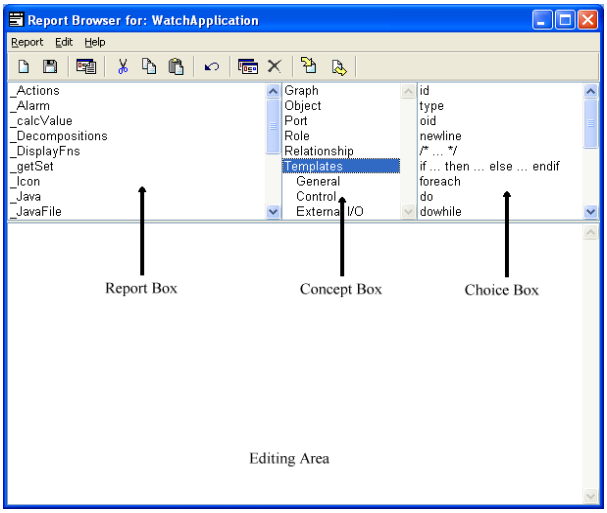

Figure 5-14. The Report Browser.

To create a new blank report definition, select **Report | New…** in the Report Browser and enter 'My HTML report' as the name for the new definition. MetaEdit+ now shows the basic definition template on the editing area. Modify the definition to look like this:

```
Report 'My HTML report'

foreach .()
{
      id; newline;
}

endreport
```

Save the modified definition by selecting **Report | Save**. Your report now appears in the list on top-left corner of Report Browser window. Make sure that the report is selected and select **Report | Run…**. You are now prompted to select the graph from which the report is generated. Choose your Family Tree diagram and press **OK**. MetaEdit+ now generates the report output that should appear as in Figure 5-15 (if there are errors during the report generation, please check that you have typed the definition correctly).

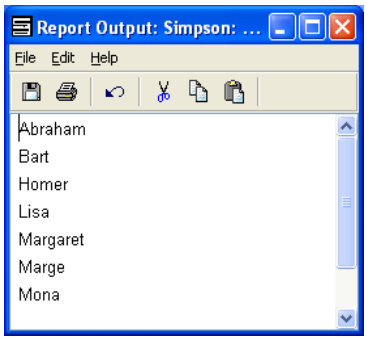

Figure 5-15. The results of running our own report.

What this definition of ours did, was to loop through all objects in a graph (foreach .()), print their identifying property – *First name* in this case – (id) and put a carriage return at the end of each line (newline). This is the basic structure of our reports: we loop through the concepts we have presented in our diagrams, extract information from them and print it out.

Printing out just the names of the members of our Family Tree is hardly a good reason to write a report of your own. But a report presenting the members of the family and their parent/child relationships in a list form, on the other hand, would be useful for various purposes. Let us modify our report definition to do this. What we need to do is to check for each person whether he or she is in either Parent or Child role (or in both) in a relationship and if such roles exist, print out the information from the other end of the relationship. Modify your report definition as follows (you don't have to enter the comment texts that are presented between /* and */):

```
Report 'My HTML report'

foreach .()
{
      /* Print out name of person */
      'Person: '; id; newline;

      /* Print out the parents */
      '   Parents: ';
      do ~Child>Family~Parent.()
      {
            id; ' ';
      }
      newline;

      /* Print out the children */
      '   Children: ';
      do ~Parent>Family~Child.()
      {
            id; ' ';
      }
      newline;
}

endreport
```

Save the report again (**Report | Save**) and run it (**Report | Run…**). You should now get a list of *Persons* with information about their parents and children. The roles (~Parent and ~Child) and relationship (>Family) of the current object are followed to its related objects by the 'do' loop.

The next exercise will be the big one. We are going to write a definition that produces a web page with a picture of our Family Tree diagram and information about each person. In the picture you can click on any *Person* to get his or her information, including the information about parents and children. Furthermore, parents and children have been linked to their respective *Person* information entries. The report also automatically opens the page in your browser. The definition that does all this looks like this:

```
Report 'My HTML report'

/* Open a HTML file for output */
filename;
subreport; '_default directory'; run;
:Family name; '.html';
write;

/* Create HTML header tags */
'<html>'; newline;
'<head><title>The '; :Family name;
' Family Tree</title></head>'; newline;

/* Create HTML document body */
'<body>'; newline;
'<h1>The '; :Family name;
' Family Tree</h1>';

/* Create picture (with image map) */
filename;
subreport; '_default directory'; run;
oid; '.gif';
print;
'<img src="'; oid;
'.gif" border=0 usemap="#';
subreport; '_default directory'; run;
oid; '.gif">'; newline;
'<br><br><hr>'; newline;

/* Generate Person entries */
foreach .()
{
     /* The name */
     '<a name='; oid; '><h3>'; id; ' ';
     :Family name; '</h3>'; newline;

     /* Date of birth */
     'Date of birth: '; :Date of birth;
     '<br><br>'; newline;

     /* Date of death */
     'Date of death: '; :Date of death;
     '<br><br>'; newline;


     /* The parents */
     'Parents: ';
     do ~Child>Family~Parent.()
     {
          '<a href=#'; oid; '>'; id;
          ' '; :Family name; '</a> ';
     }
     newline; '<br><br>'; newline;
```

```
        /* The children */
        'Children: ';
        do ~Parent>Family~Child.()
        {
              '<a href=#'; oid; '>'; id;
              ' '; :Family name; '</a> ';
        }
        newline; '<br><br><hr>'; newline;
}

/* Create HTML footer */
'</body>'; newline;
'</html>'; newline;

/* Close the output file */
close;

/* Launch the web browser */
external;'"';
subreport; '_default directory'; run;
:Family name;'.html"';
execute;

endreport
```

Save the report again and run it. Try out the web page that was generated. As you see, it's basically just a simpler version of the web page we generated with the predefined report earlier. However, there is one important difference: the new web page demonstrates the domain concepts better than the old one and this is exactly what we wanted in the first place. General, predefined reports can be useful, but creating your own domain-specific reports is the key to getting the maximum gain out of your domain-specific modeling language.

5.4 FURTHER IDEAS FOR IMPROVING THE FAMILY TREE MODELING LANGUAGE

We have now completed our second development cycle of the Family Tree modeling language. Even though this could be the first version we could use for some real modeling, it is very usual that the modeling language continues to evolve. The next most probable target of development in our Family Tree language is the *Family* relationship. Though we are not going to explore it further within this tutorial, here is some food for thought in case you want to experiment with it by yourself.

Currently the *Family* relationship in our Family Tree modeling language is semantically quite loose. This is both an advantage and a disadvantage. As it just presents the situation where a *Male* and a *Female* can have a union that produce children, we can model all such situations easily with the same relationship type (e.g. there is no requirement for parents to be married before they can have children). The emphasis here is on natural bloodline. However, in many cases we are actually not so much interested in the bloodline as the family as a legal entity. For instance, we may want to separate families where parents are married from those where

parents are not married. This may require a specialization of the *Family* relationship in the same way we did with *Male* and *Female*.

Another important question is the status of adopted children. In the family tree that presents only bloodline relations they are not shown, but in legal family trees they may have an important role (e.g. in the families of Roman emperors an adopted child could become an heir to the throne). Perhaps the best way to do this would be to specialize the *Child* role as *Physical child* and *Adopted child*.

A further step would be to apply the Family Tree modeling language – with modifications of course – to some other problem domain. One possible domain for such an application could be animal breeding. The people who breed animals like horses and dogs need to store information about their animals. They also want to be able to follow and analyze certain properties of animals through the breeding lines and generations. This would require a whole new set of properties to be included into the Family Tree modeling language – or is it a Breeding Line modeling language already?

As you can see, the need for modeling language constructs and how they are implemented originates from the problem domain itself and the intended use of the language. Finding the required domain concepts is a very important part of creating a modeling language and also one of the more difficult tasks to carry out. However, MetaEdit+ as a tool provides you with a solid and flexible basis, so do not hesitate to experiment with your ideas!

6 The Conclusion

We have now walked through a set of examples of how MetaEdit+ can be employed: first as a tool for creating your own domain-specific modeling language, and then as a tool supporting that language. The ultimate question after all of this of course is what does it have to do with your organization? What kinds of benefits can you look forward to by adopting the approach presented here?

According to our experiences and those of our customers, the three key benefits from adopting metaCASE technology are:

1) **Productivity increases by as much as a factor of 10**. Traditionally software development has required several error-prone mappings from one set of concepts to another: first from the domain concepts in requirements documents to design concepts (like UML), and then from design concepts to programming language concepts (like C++). This is equivalent to solving the same problem over and over again. With domain specific modeling language, the problem is solved only once by working with pure domain concepts. There is no need for error-prone mappings of concepts as the final products are automatically generated from these high-level specifications. Studies have shown that this kind of approach is 5-10 faster than the usual current practices.

2) **Better flexibility and response to changes**. Focusing on design rather than code results in a faster response to requests for changes. It is easier to make the changes at the domain concept level, and then let the tool generate code for multiple platforms and product variants from a single set of domain concepts.

3) **Expertise shared with the whole development team**. The usual problem within development teams is the lack of expertise among the developers in turning domain ideas into good code. It takes a long time for a new developer to learn enough to become productive. Also, even the more advanced developers need to consult with domain experts frequently. In the approach presented in this tutorial, the expert defines the domain concepts, rules and mapping to code. Developers then make models with the concepts guided by the rules, and code is automatically generated. The code is thus being produced with the expertise of the expert developer, so quality improves significantly.

Obviously there is more to this than is contained in just this tutorial and the key benefits we have listed here. To learn more about adopting this kind of approach and to see a more comprehensive example of a domain-specific modeling language, you are encouraged to study our Watch modeling language example. If you are interested in knowing how others have utilized this technology, our web pages at http://www.metacase.com provide a good starting point. There you will find user references, downloadable white papers and FAQs. If you have any questions about metaCASE technology, or are thinking about how your organization could benefit from it, do not hesitate to contact us!