Version 4.5

The S60 Phone Example

MetaCase Document No. SE-4.5

First Printing, 2nd Edition, February 2008.

MetaCase
Ylistönmäentie 31
FI–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 S60 phone example illustrates how cellular phone applications can be modeled and generated based on DSM. To achieve this, a domain-specific modeling language is implemented into MetaEdit+ along with a generator for producing code. Using the modeling language, a developer can design phone applications using directly the domain concepts of the phone, like its widgets and services. Generators are used to produce the executable code, automate the deployment to run the code in an emulator (with a single-click), check the models and produce documentation.

This example covers both the issues related to S60 application modeling as well as in part how the DSM was made. First, we inspect the modeling language with some examples and then we discuss the issues of modeling language and generator specification. Compared to other DSM creation examples, we will focus on showing how to import external symbol elements in order to define notation as well as ports. On the code generation side, special focus will be put to integrating library code and manually written code with the models.

For exploring the S60 phone example thoroughly, the following things are required:

- MetaEdit+ for trying out the S60 phone language. The phone example can be found from the demo repository, from the project named 'S60 phone'. For further information about MetaEdit+, please refer to the MetaEdit+ User's Guide.
- Nokia S60 SDK emulator for running the generated applications. The emulator you can download from www.forum.nokia.com. Download S60 SDK which is named as "2nd Ed, FP2 146 268". There may also be newer versions available. Alternatively you can use a real target device, Symbian smartphone using S60 UI framework.
- Python-interpreter for running Python code in the emulator or in a real target device. This Python package can also be downloaded from www.forum.nokia.com. Download a package named "Python for Series 60 1.2 for Series 60 2nd Edition". This should download a file named PythonForSeries60_1_2_for_2ndEd_FP2_SDK.zip. Before installing it you must first install S60 SDK. We recommend that you install them using the default directories as suggested by the installers.

We expect that you have knowledge about using MetaEdit+. If you want to extend the DSM further − add notational symbols, additional constraints, generators or by modifying dialogs and toolbars for the modeling tool − you should have MetaEdit+ Workbench or the evaluation version available from http://www.metacase.com.

1 The S60 phone example

The S60 phone example presents a DSM language and its tool support, specifically tailored for developing applications into cellular phones. The target for the code generation is Python for Series 60, a framework which runs on Symbian smartphones. Because of this target, the modeling language is based on the architecture, phone services and widgets that this particular framework provides. Naturally, it would also be possible to model and generate code for other cellular phone frameworks in a similar way. Actually inside S60 already two other frameworks and programming languages (C++ and Java) are supported in addition to Python. DSM for these is also implemented, but they are outside the scope of this S60 phone example.

In this first chapter we introduce the S60 modeling language and its usage scenarios. Chapter 2 continues to explain how to use the language with an example: we modify an application design and generate code to run the modified application. Chapter 3 describes how the language and generator were defined. Rather than focusing on basic metamodeling capabilities that are discussed in the tutorial examples on metamodeling, here we inspect selected aspects of language and generation creation. On the language side we show how to define ports for representational purposes and how to use external graphics for the notation. On the code generation side, we show different ways to refer to manually written code from models.

Please note that testing the modeling language and models presented here requires basic knowledge on how to use MetaEdit+.

1.1 THE BASIC IDEA OF THE DSM FOR S60 PHONE

The general objective of the "S60 phone language" is to ease and speed up application development. This is achieved by raising the level of abstraction directly from programming concepts to UI concepts and phone services. Doing this hides the unnecessary complexity as application developers do not need to master the details of the phone architecture and related programming model. Briefly, by using the modeling language a developer draws a graphical design of the application logic using S60's UI elements (lists, forms etc.) and services (SMS, accessing files, taking a picture etc). At any stage of the design phase, a developer can run a generator that produces application code ready for execution. The generated code uses the API services provided by Python for S60 framework.

This domain-specific language pushes application design toward extreme simplicity and easiness by using high-level UI concepts, widget visualization that is 1:1 to the actual phone, and by drawing behavioral logic with arrowed lines. Furthermore, the modeling language covers architectural rules that prevent developers to make designs that are illegal: ideally, if a developer can draw the application, it will work.

1.2 AN EXAMPLE

Using the DSM for S60 phones, the design process goes as follows: A developer specifies the application by thinking about the wanted services of the phone, the UI that is needed and navigation flows within the application. These phone-dedicated concepts are directly the

modeling language's concepts too. They can be selected from the editor toolbar and placed on the drawing area. The modeling elements can be further specified with related properties and connected together to specify the navigation flows.

An example of an application design is illustrated in Figure 1-1. If you are familiar with some phone applications, like phone book or calendar, you most likely already understood what the application does. It allows a phone user to register for a conference via text messaging, view the conference program and speaker data or browse the conference program via the web.

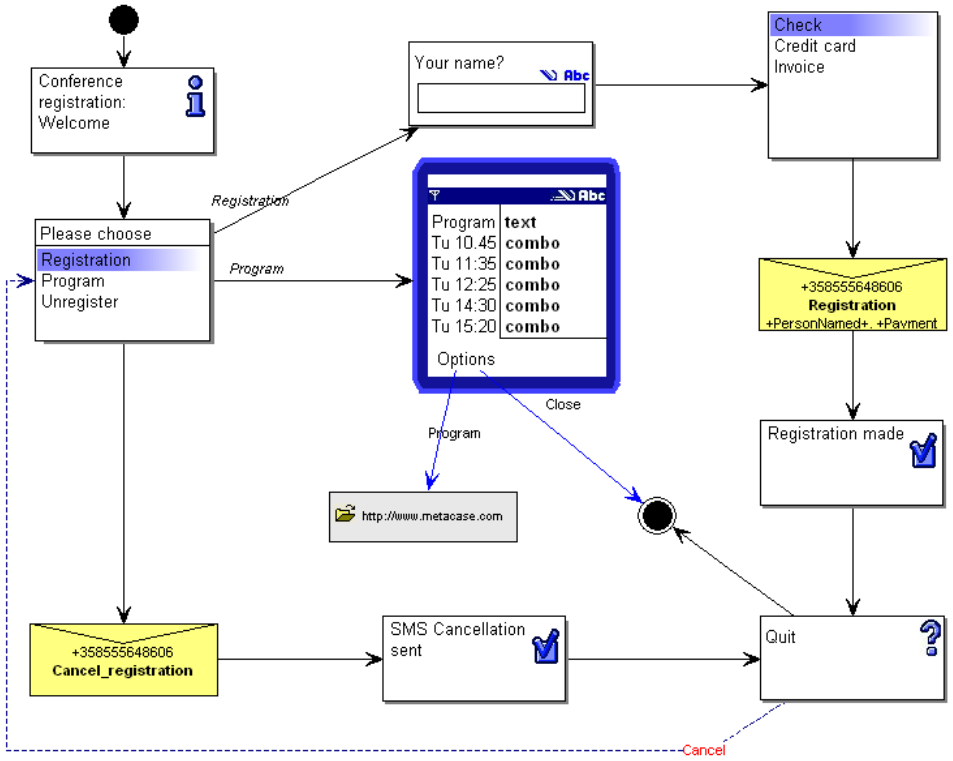

Figure 1-1. Conference registration application

As you can see from the model, all the implementation concepts are hidden (and are not even necessary to know at this stage). Developers can focus on finding the solution by using the phone domain concepts, such as Note, List, Form, SMS sending and Query.

The modeling language covers also phone domain rules, which prevent developers from making illegal designs. For example, in S60, it is typical that after sending an SMS message, only one UI element or phone service can be triggered. Accordingly, the DSM allows only one flow from the SMS element. This means that application developers do not need to master the details of the S60 architecture and programming model. If you understand the phone UI and services it can provide, you can start developing cellular phone applications.

Finally, a developer can run a generator to produce code and execute the application in an emulator. If you have not installed the SDK and Python Framework, you may still inspect the generated code. Figure 1-2 illustrates how the application is executed in an emulator. This application is generated from the model illustrated in Figure 1-1. By default the generator expects that the emulator is installed into the default directory of the SDK installer. You may modify the target directory for producing the Python code by using the Generator Editor provided by MetaEdit+.

Figure 1-2. Running the generated code in an external PC emulator

The generated code uses the services that the S60 platform and its framework provide. After design, there is no need to map the solution to implementation concepts in code or to modify the generated code. Built applications should also have better quality since typical errors found in manual programming do not occur anymore.

In addition to generators that produce code, a developer may also produce final documentation of the application or run design checking by using specific generators.

1.3 ABOUT THE S60 PHONE MODELING LANGUAGE

The modeling concepts of the language are shown in the editor's toolbar. They can also be seen from Types menu. Figure 1-3 shows the contents of the Types menu and summarizes the modeling concepts with their notational symbols.

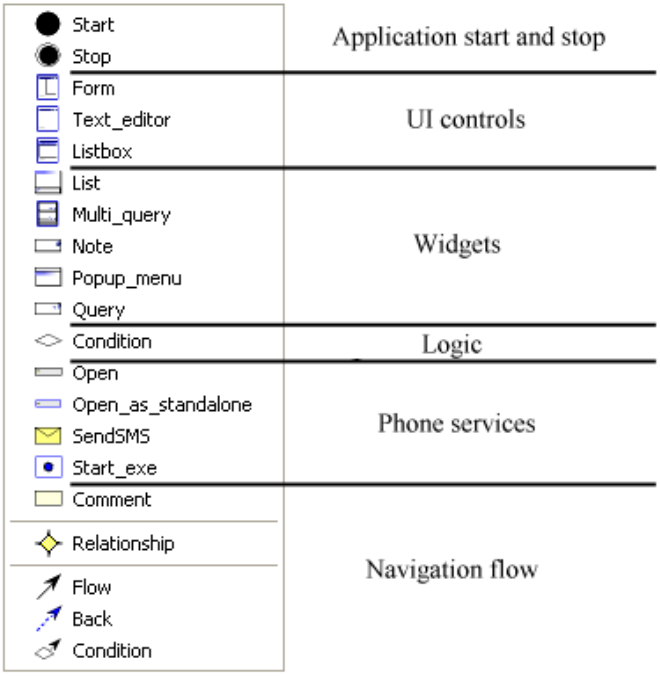

Figure 1-3. Modeling concepts for S60 phones

The modeling concepts are grouped as follows:

- 'Start' and 'Stop' concepts are used to specify application start and end states. Their semantics are close to Start and End states as found in state machines.
- UI controls (Form, Text editor and Listbox) fill the whole display. They have their own state behavior and richer internal structure than other UI elements.
- Widgets represent the different dialogs that are available for the application.
- 'Condition' is used to specify logic and extra rules for the navigation flows.
- Phone services represent the concepts that access the Symbian and S60 phone services via the Python API. These include accessing files stored in the phone, browsing web, sending SMS (short messages), making calls and accessing other pre-built phone applications, such as calculator, calendar or camera.
- Navigation flow is specified by using three different kinds of relationships:
    - 'Flow', which describes the mandatory navigation flow of the application.
    - 'Back', which is used to specify navigation canceling when the normal default cancel policy is not seen as adequate.
    - 'Condition', which is used when application logic requires specifying conditions that could not be described by using the other modeling concepts.
- 'Comment' element is used to attach free textual descriptions as comments that are visible in the design model.

2 Working with the DSM for S60 phone

In this chapter, we discuss how to access the example in MetaEdit+ and how to work with it: First by playing around with an existing application design and then by modifying the application using the modeling language. Finally, we generate the modified application and run it in a PC-based emulator.

2.1 ACCESSING THE S60 PHONE EXAMPLE

To access the S60 phone example, start MetaEdit+, choose 'S60 phone' from the project list and login as usual into the demo repository. When MetaEdit+ has completed the login procedure, the S60 phone example can be accessed with the normal MetaEdit+ browsing and modeling tools like the Graph Browser and the Diagram Editor.

2.2 PLAYING AROUND WITH THE S60 PHONE LANGUAGE

We start by inspecting already available design models. They are listed in the Graph Browser in the main MetaEdit+ window. Double-click the graph named 'Conference registration: Application' to open it in Diagram Editor. This application is already discussed in Section 1.2.

The application logic is drawn by using flow relationships from one start state to stop states. You may inspect the properties of any model element by double-clicking them in the diagram or in the Diagram Editor sidebar. You may also access operations related to each model item by first choosing the element and then opening its pop-up menu.

2.3 MODELING TO CHANGE THE APPLICATION

The next step in working with the modeling language is to change the current application logic by adding new elements to the model and changing the navigation flows. The change request for our modeling tasks is as follows:

a) If the user wants to stop the registration process, control must go back to the start menu for reselecting the required action.

b) If the user wishes to cancel the registration, the application has to ask for a confirmation prior to sending the SMS message.

c) Search functionally for choosing a payment method is unnecessary as there are few options only.

Let's next model the application accordingly and then generate code to execute the modified application to verify the changes. The next sections show how to make the changes into the model.

2.3.1 Adding a new relationship for the navigation flow

The default policy to navigate backwards is to go to a previous UI element; excluding elements that made irreversible actions like sending text messages, making a phone call or taking a picture with the camera. In our current design model pressing cancel while choosing a payment selection follows this default policy: it returns to the Query that asks the user to fill out a name. This cancel behavior is not represented in the model as it is default behavior and therefore not needed to model. Now, however, we need to specify that the application returns back to the start menu when cancel is pressed instead of entering registrants name.

To change the default policy, we need to add a new back navigation flow. Since we already know the flow type, named in the language as 'Back', the fastest way to create a relationship is to choose the 'Back' relationship type from the toolbar, followed by clicking the elements that we wish to connect.

To add a breakpoint as an angle of the relationship line you may click the drawing area in the places where you would like to add a breakpoint. One such option is illustrated in Figure 2-1. You may also move the relationship and/or add breakpoints to the line for better routing afterwards.

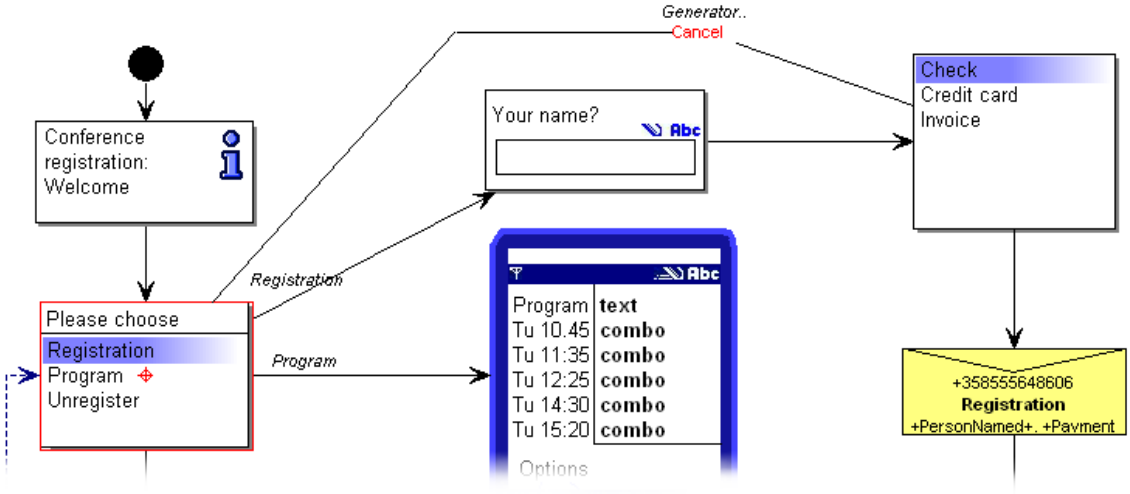

Figure 2-1. Adding a cancel navigation relationship to the application

After connecting the elements a dialog is opened to specify the properties for the Back flow. In our case they are not needed so just press **All OK**. Now the design is updated and we could generate the code and test the application by pressing the 'Build' button in Diagram Editor's toolbar.

2.3.2 Adding a new widget element

The second request was to ask for a confirmation before sending a message for canceling the registration. Here we need to add a new dialog, which asks the user to confirm. For this purpose, the 'Query' object is most suitable. To add a query, press the 'Query' button from the toolbar or select it from the Types menu. Then click in the diagram into an empty place to add a new Query. This opens a dialog where we can specify the details of the element.

For a 'Query' we can enter its 'Prompt' text (like 'Do you want to unregister?') and fill other needed properties. Here only the query type is important: Choose 'query' from the list to have a Boolean query. The Boolean query means that pressing OK in the application continues the flow and pressing Cancel goes one step back. Other property fields are not necessary: 'Internal name' allows to give a name for the query that will then be shown in the generated code

(otherwise the generator gives a name), 'Initial value' is not relevant as the selection is made directly by using the navigation buttons, and entering 'Return variable' stores the value into the named variable. In the conference application such information is not needed, unlike for example when sending the registration message which sends as arguments data stored in variables for payment method and registrant name.

After you have defined the two relevant properties, the dialog for specifying a query should look like in Figure 2-2. Choose **OK** and close the dialog. This adds the created object to the diagram.

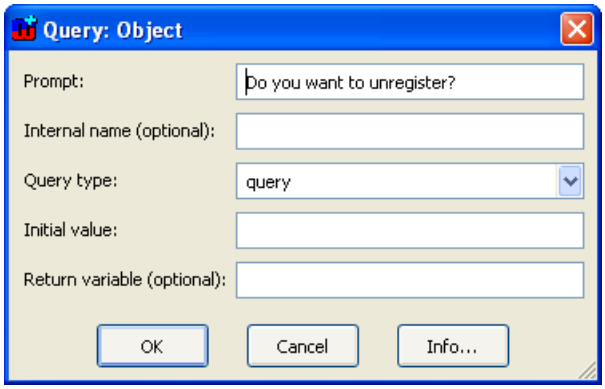

Figure 2-2. Adding a query to the application design

Next, you can delete the current relationship from the Popup menu to the 'Cancel_registration' SMS message. Then move the created query between the Popup menu and the 'Cancel Registration' SMS sending object and connect it with them. Relationships can be created in many ways, but let's use here one which is not based on pre-selecting a relationship. First, click the Popup menu object in the diagram that asks the user to choose the operation (of 'register', 'program' or 'unregister'). Then open its pop-up menu and choose **Connect**.... Next click the query element you created to specify the target for the relationship. This opens a dialog asking a relationship type. As we did not pre-select a relationship from the toolbar MetaEdit+ asks to choose among possible ones. Double-click the 'Flow' or alternatively choose it and press **OK**.

This creates a relationship in the model. If the relationship or its roles have properties, a new dialog opens allowing their entry. In our case we can specify the value that is chosen in the pop-up menu to follow the unregistration path.

Click the tab 'From choice: Please choose' and enter the choice value as illustrated in Figure 2-3 below and press **All OK** in the dialog.

→ *Strictly speaking, the choice value is not mandatory here because other possible choices are already specified. The code generator thus expects that if other values are not met, the one with no choices will be followed. If more than one choice value is left unspecified, a model checking will report a warning and points out data that need to be specified.*

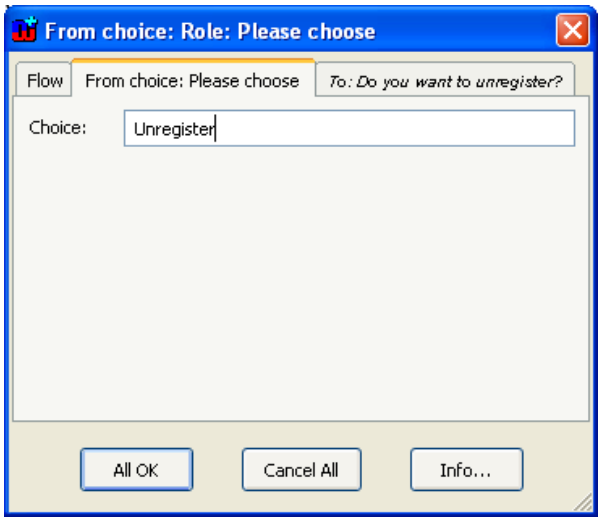

Figure 2-3. Entering a choice value for the navigation path

Finalize the navigation flow by connecting the created query to the SMS element sending the 'cancel_registration' message. You can create the relationship as done earlier or choose first 'Flow' relationship from the toolbar and then click the elements in the navigation order: first the query as a source and then the SMS sending as a target.

2.3.3 Changing a model element

Finally, we need to remove the unnecessary search field from the list element that provides the payment options: as there are just three payment options, the provided search functionality for long lists by using the phone keyboard is not needed here. To change the design model accordingly, you need to modify just one property of the List object. First double-click the List object for choosing payment method and then select the 'Search field enabled' property (see Figure 2-4).

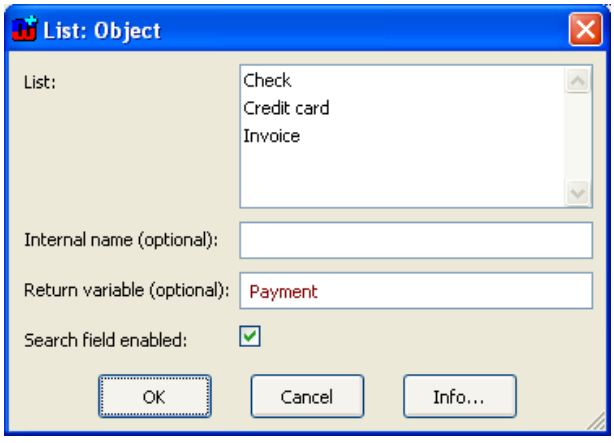

Figure 2-4. Modifying properties of a design element

After the modifications the application design should look like in Figure 2-5.

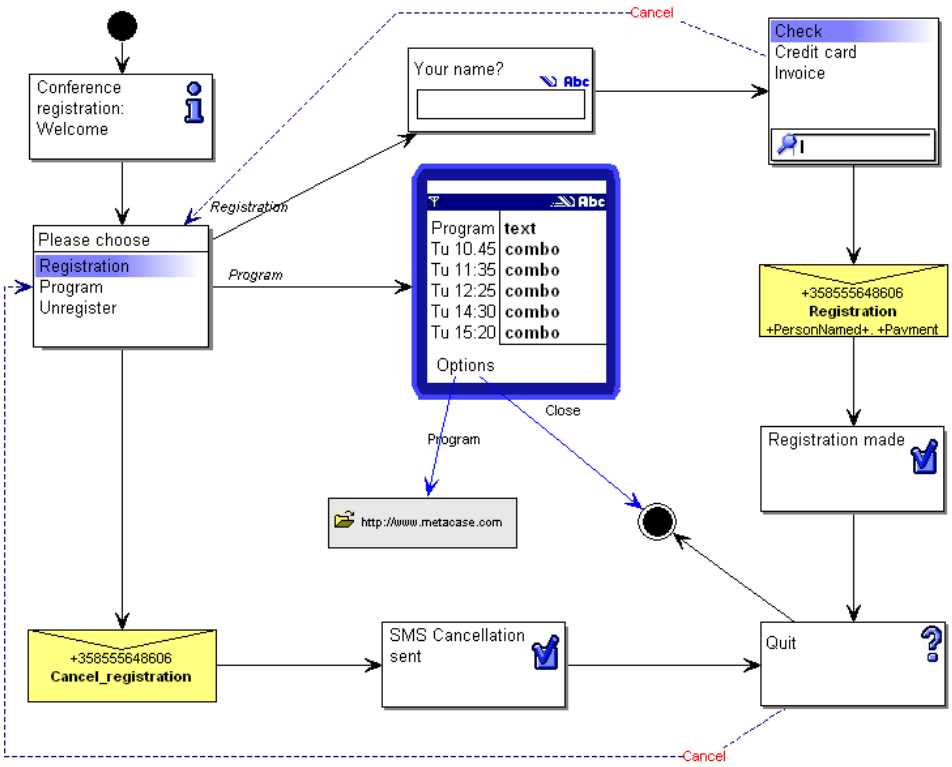

Figure 2-5. Modified conference registration application

2.3.4 Generating the application code and running it in an emulator

The DSM is made to support agile development: At any stage we could have produced the code and tested the application by simply running the generator. After making all three changes, we can now generate code to run the application in a PC emulator. If the emulator is not installed (see instructions in the preface) then you may only inspect the generated code.

To execute the application simply press the 'Build' button in Diagram Editor's toolbar. This produces the code and starts the emulator so you can run the generated application. After the PC emulator starts, choose Python from the phone desktop (you may need to scroll down to see the Python icon). Then press the **Options** key and choose **Run script** from the menu. This opens a list of available applications. Choose here 'Conference registration' and press **OK**. You may now use the application to test the changes you made. Figure 2-6 illustrates the query added.

→ *In the emulator, the SMS sending does not work as it does not have access to a cellular network. The generated applications show instead a note that displays the content of the SMS message. For deployment in the product, you can modify the SMS sending part of the generator and remove the note code and comments from the actual SMS sending message. See code generator definition for details.*

Figure 2-6. Running the modified application after code generation

You may also run other generators, like those that produce documentation, run metrics and model checking. You are also welcome to make other changes to the application, inspect other available design models or to create totally new applications – from model to executable code. We leave this part to you and move next to DSM implementation.

3 Creating the DSM in MetaEdit+

We have now used the S60 phone language. Next, we shift to the language and code generation definition.

3.1 LANGUAGE DEFINITION

MetaEdit+ Workbench User's Guide describes the functionalities for defining modeling language in more detail. These metamodeling functionalities are also discussed in tutorials, such as the Evaluation Tutorial, the Watch Example and the Graphical Metamodeling Example. We focus here on two special aspects of language definition: importing external graphics to the language's notation and using ports.

3.1.1 Importing external graphics into language symbols

The DSM for S60 phone is based on following 1:1 visualization to the actual phone. The notational symbols used in phone UI can be thus applied directly in the modeling language. MetaEdit+ allows importing the external graphical elements either as bitmaps (BMP, GIF, JPG, PNG) or vector graphics (SVG). Next, we show one scenario on how to import graphics as symbols to represent a modeling concept and give conditions for its visualization.

In the S60, each query instance is visualized with a dedicated icon to show which kind of query we have. For example a question mark is used for the Boolean type of query. This symbol, available as a bitmap, can be imported with Symbol Editor by choosing **Symbol | Import Bitmap…** and then selecting the respective file.

Once the symbol element is added, its representation can be made conditional based on the values given by a modeler. In the case of the question mark it should only be shown when the modeler selects 'query' as value for the 'query type' property. To add such a constraint, select the question mark symbol in the Symbol Editor and open its formatting setting by choosing **Format…** from the pop-up menu. Then go to the condition tab (see Figure 3-1) to specify the condition.

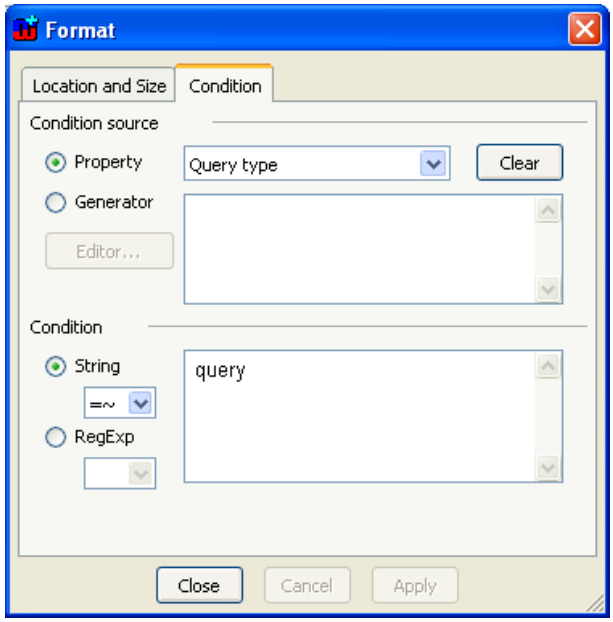

Figure 3-1. Specifying symbol condition

Symbol conditions are specified by selecting first the source and then giving a value for the condition. The condition source can be based on a property or on the output of a generator. A generator-based condition source allows you to specify almost any condition for symbol visualization. Often, like here, the condition is based on one property only.

As shown in Figure 3-1 a property is chosen to be the source and from the list 'Query type' is selected. In the condition part, the value the condition must meet is specified to be a 'query'. This is the same value as specified in the metamodel. As a result, when a query type is chosen to be a Boolean query in the model a Question mark is shown. Respectively those symbol elements of Query whose conditions are not met are not shown.

3.1.2 Defining ports

The S60 language uses ports as representational elements so that main UI controls like Form can be related to other elements via two kinds of connections. UI controls have also own menus and their specification in the language is implemented using a port. If you select a form in the model you can see that the symbol has two connectables from which relationships can be drawn: a larger main symbol and a smaller for Options menu.

To see how the representational port is defined open the Symbol Editor for the Form object (see Figure 3-2). Then choose the connectable shown as a red rectangle related top the Options menu. The connectable is selected when its target point in the middle is selected too. If the target point is not selected, then you may have chosen the Options text element. The Symbol Editor shows the chosen object in the active field of the status bar.

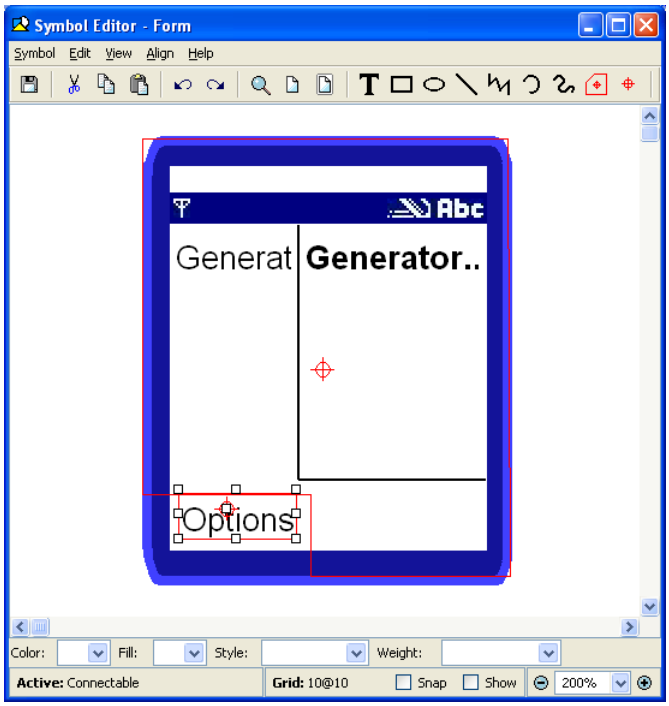

Figure 3-2. Form symbol

Open the **Format…** menu from the pop-up menu to see the port definition given for the selected connectable. Figure 3-3 shows that the connectable uses port 'Left softkey'. As pressing the left softkey in the real S60 phone opens the Options menu, the port is named accordingly.

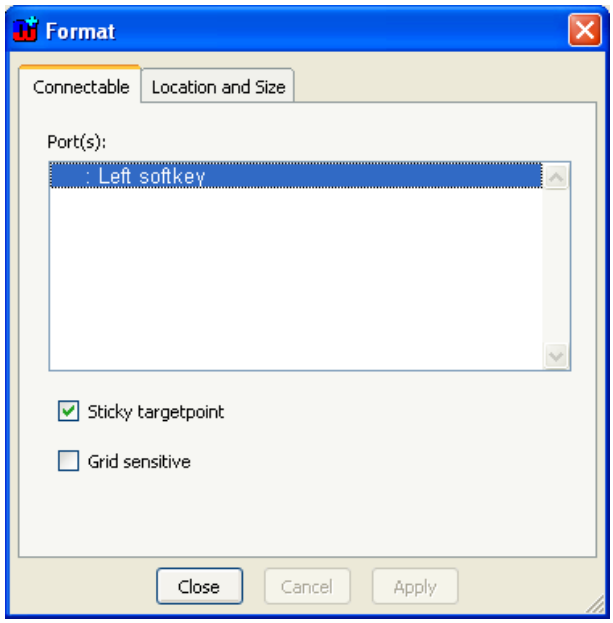

Figure 3-3. Relating ports to connectable

Ports are defined with the Port Tool similarly to other language concepts. In this case, the port is just used for representational purposes so the 'Left softkey' port does not have any properties. For details on Ports see MetaEdit+ Workbench User's Guide.

3.2 CODE GENERATION: RELATING MODELS AND CODE

It does not always make sense to generate all code directly from models. Companies often have already existing code or the DSM could not be made to specify all the functionality using domain concepts. In this section we represent some approaches by using the S60 example to integrate models with manually written code.

3.2.1 Generator provides integration with the component library

Usually, the best approach is to create a component library or a component framework that provides prefabricated software building blocks that can be used when developing applications. This was exactly the case of S60 as its services are made available via the Python framework. The code generator then calls directly the services provided by this framework. Creation of such a component library does not necessarily require a lot of effort as such components may already exist from earlier development efforts and products, and just need a little tweaking.

To illustrate how a code generator calls the services of the framework, let's highlight the case of Query. You can see the code generator for the Query object by selecting **Graph | Edit Generators…** in the Diagram Editor. This opens Generator Editor. Choose the generator named '_Query' from the list on the left. You will find it in the tree structure under '!Build'. Your view in Generator Editor should then look like in Figure 3-4. It shows the complete generator definition for producing Query code.

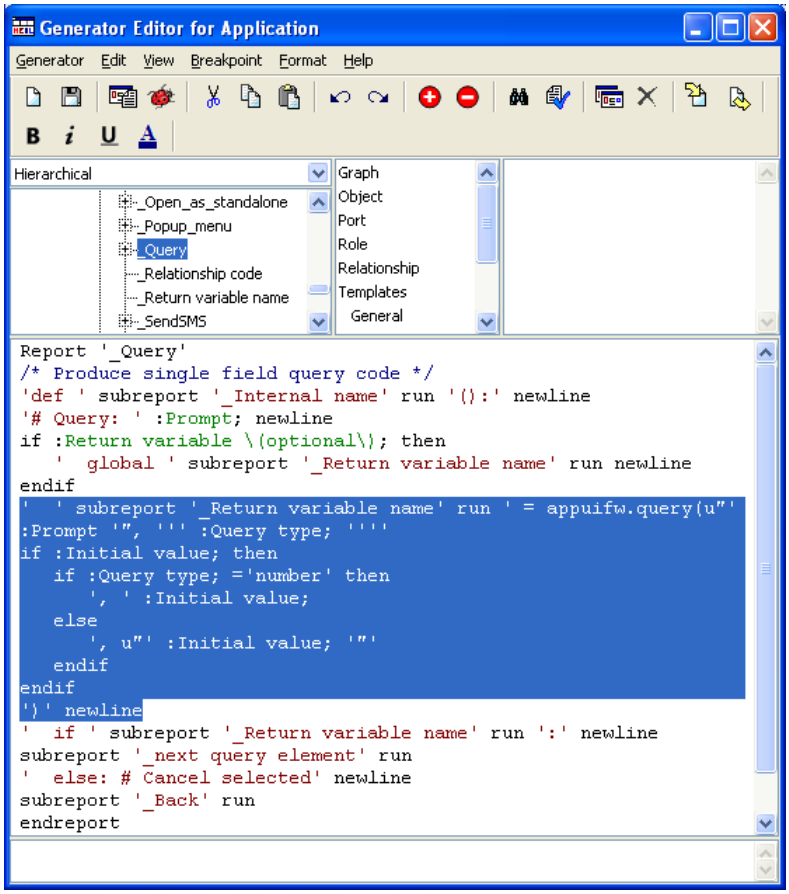

Figure 3-4. Calling the service of the underlying framework

The highlighted area in the figure shows the part of the generator that makes the actual call to the Python framework. In other words, it takes the data from the model to produce the call to the API. Code produced by the highlighted generation definition could look like:

```
PersonNamed = appuifw.query(u"Your name?", 'text')
```

This code is actually taken from the Conference registration application where a query element is used for entering registrants name (see Figure 1-1). The first highlighted line of the generator creates the variable in which the query value is saved, followed by the call to the framework, i.e. to appuifw module and its query service. The rest of the highlighted generator fills the needed parameters by taking them from the models. Actually, the second highlighted line creates already the needed two parameters for the sample code shown above: the prompt text and the type of query (i.e. 'text'). The rest of the generator includes the initial value to the call if such is specified in the model.

As this small example illustrates, the DSM raises the abstraction and hides the complexity as a developer who uses the modeling language does not need to know about Python, learn and master the phone framework or know the programming model when making the applications.

3.2.2 Referring to libraries from the models

The S60 language also allows specifying Python code directly in the models. The language is thus extended from pure domain concepts to include code concepts too. The places for using code, however, are restricted to only a few places: those where it makes sense to apply a code library code or write code manually. Also, the code concepts in the modeling language are restricted into two: referring to library code or writing code manually.

Consider the calculation example shown in Figure 3-5. This model is also available in S60 project. This small application asks for two number values and shows their sum in a note dialog. The calculation algorithm is not specified by using the modeling concepts but rather entered as a textual specification into the note. We use here Python for writing the algorithm as the generator can take this textual specification and use it directly in the generated code.

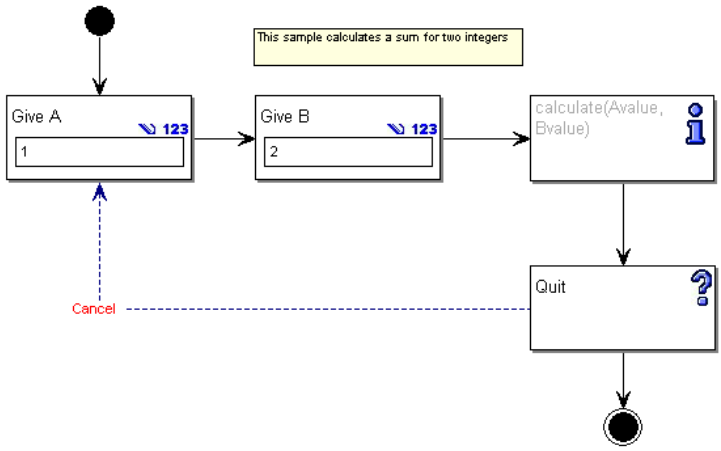

Figure 3-5. A design model referring to a code library

Double-click the note element to see its properties. Rather than showing text inside the Note element its content is now treated as a code. The actual code comes from a function library and a function 'calculate' is chosen. Avalue and Bvalue are then provided as arguments to the function. Note that these two values are defined as return values for the two queries made before the calculation.

Libraries are usually considered to be either black-box or white-box. In the case of black-box we see just the interface (here: 'calculate (a,b)') but not its implementation. For the S60 case the language is defined to be white-box: contents of the library function can be seen and modified. Double-click the Function used property 'calculate (a,b)' to see its implementation. Figure 3-6 shows the implementation and description of the calculate function.

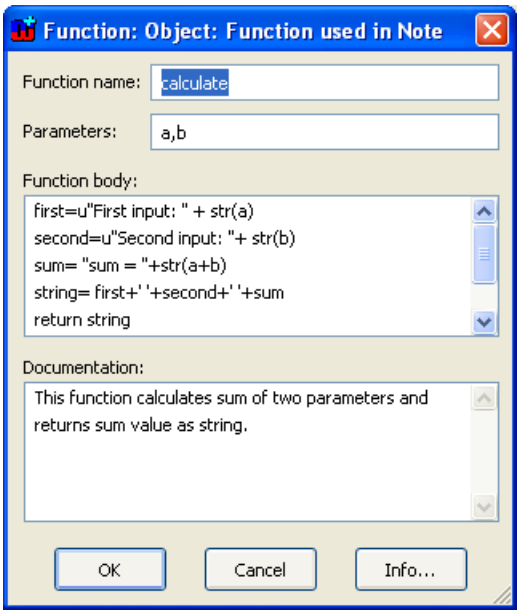

Figure 3-6. Selecting a library function

The S60 language allows defining functions while modeling as well as importing them from an existing library into MetaEdit+ by using the API functionality. Such functions can be imported completely or you may choose to just making their public interface available in models. In the case of S60 language, the code generator expects that Functions are imported completely, so their content is also generated into the application code. Alternatively, a code generator can import the function from the library during generation.

3.2.3 Entering code directly in models

Instead of referring to or importing libraries, it is always possible to have one or more language elements just for writing code directly into the models. This is obviously not as good an option as having a framework or libraries, but in selected places it can be useful. In the case of the S60 language, a Form has for instance an optional 'save validation' function that is run to check that the changes made into a Form are correct. In the modeling language such 'save validation' function is entered into a textual property.

For a more complex case let's look at the condition element in the S60 language. Condition allows specifying additional rules, checks and other navigation information that goes beyond the direct DSM support. In the 'RestaurantFinder' application (see Figure 3-7) a condition is used to check that the given zip code is legal. In our case, it means that a zip code has 5 digits.

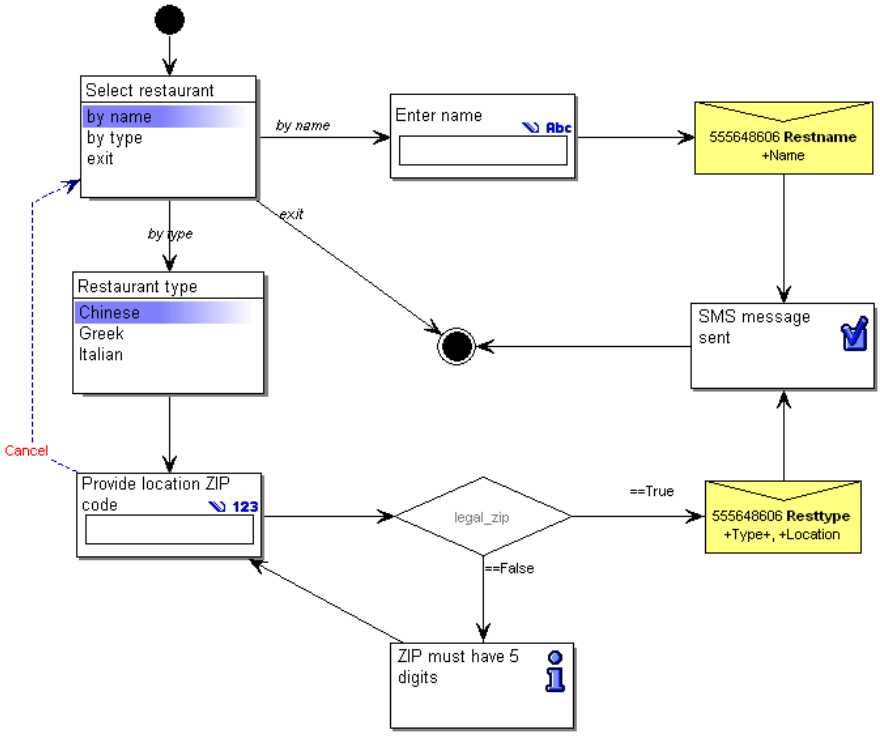

Figure 3-7. Restaurant finder application with a condition for ZIP code validation

For the checking purpose, a condition named ‘legal_zip’ is added to the diagram. Double-click it to see its details as shown in Figure 3-8. The ‘Condition’ object has two properties: a ‘Condition variable’ and ‘Condition’. The code is entered as a Python script so that it can be used directly by the code generator. The code returns either True or False. This value is then used for guiding the navigation: if the length of the zip code is 5 characters a SMS message is sent, otherwise a note is opened informing that ‘ZIP must have 5 digits’ before going back to the ZIP code entry.

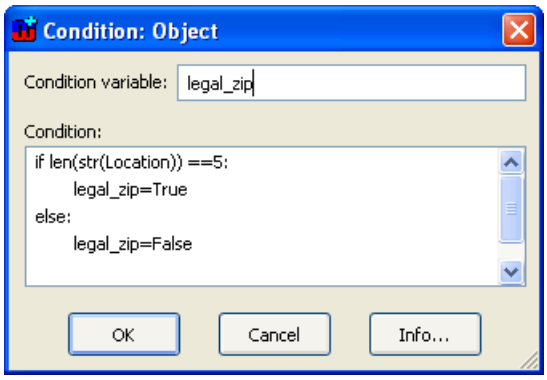

Figure 3-8. Restaurant finder application with a condition for ZIP code validation

Instead of having here code written directly, it is also possible to use library functions similar to the calculate function that we discussed earlier. This would allow having a library of common conditions that would then reused in design models.

In addition to the ‘Condition’ object the S60 language allows to write manual code also to the same places where the use of libraries is possible. If there is no reference to a library function then code is expected to be written directly into the text field. The text entry can be written using the MetaEdit+’s internal text editor or by using your favorite editor. The Options settings in the MetaEdit+ main window allow you to specify the use of external text editor. See MetaEdit+ User’s Guide for details.

3.2.4 Regeneration with protected blocks

Final option is to generate code with protected blocks: Generate partial code which can later be modified without loosing the manual changes when the code is regenerated. In the case of the S60 language such is not needed as the modeling language already identifies places for writing the code. For the sake of demonstration, however, we will add a protected block definition to our current generator so that manually written condition code can later be modified.

While defining protected blocks, we need to consider two things: identify parts of the code where manual changes are needed and define how blocks are described in the code. For the demonstration we want to generate condition code into a protected block.

→ *Better use of protected blocks would be here if the modeling language would not have a conditional code concept at all and the generator would create a template or skeleton code based on the model, like variable names, inside the protected blocks.*

Protected blocks must be defined such, that their code works according to the target language. Therefore we need to first define the how protected blocks are presented in the generated code. MetaEdit+ provides a default block definition but for our case of Python we need to specify our own. Consider the following generator definition for producing a protected block into a generated Python file.

```
filename subreport '_default directory' run id '.py'
     md5start '# MEPMD5 ' md5stop newline merge
     subreport 'Generate Python script' run
close
endreport
```

The first line of the generator specifies that the regeneration is used for a file named using the model `id` and '.py' extension for a python file. The second line defines how checksums are presented and here the only change to the default values is the use of '`#`' used for making a comment in Python.

Next, we need to modify the generators where code to be written as protected block is produced. For the condition code we must open the Generator Editor and its generator named '_Condition'. For this generator, we specify start and end for the protected block by using reserved words of `md5id` for start and `md5sum` for the end.

In this case, we want to include only one value from the model into the block: a condition code. Figure 3-9 shows the generator definition when protected block definition is added into the generator. The highlighted line starts the block with `md5id` tag, followed by the unique id within the file. Here, the `oid` refers to the unique identifier MetaEdit+ gives for all model elements. The line includes also comment text so that the block can be identified from the generated code. The 'Do' clause then takes the value from the model and indents every line to follow Python conventions to indent code inside a function. Finally, `md5sum` is used to end the protected block.

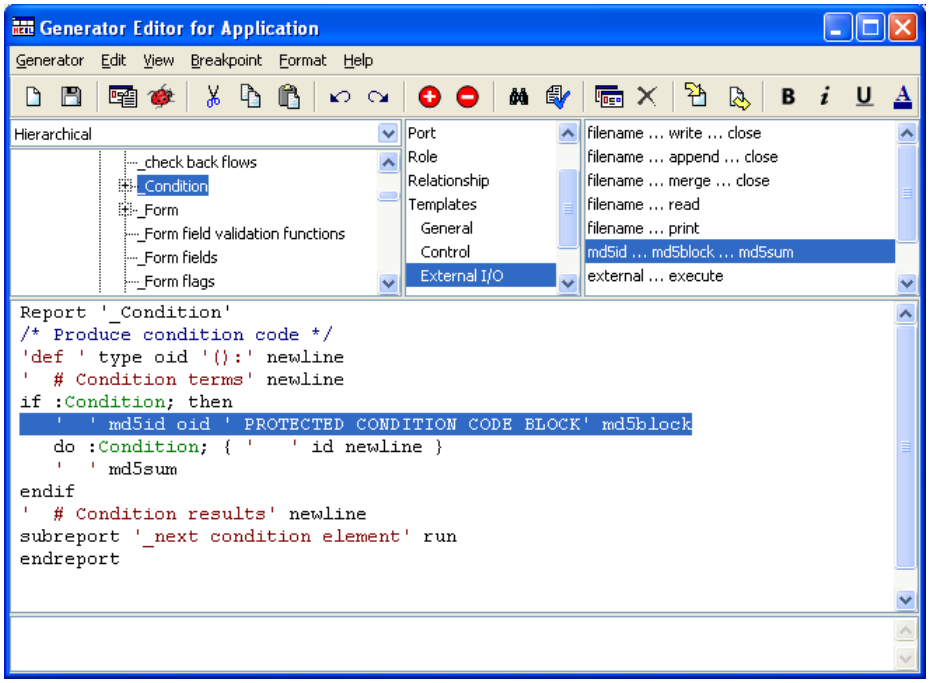

Figure 3-9. Generating code with protected blocks

Now the generator produces checksum for the block that is evaluated during regeneration. If changes are made into the block, its content is not generated. In case of the Restaurant finder application the generated code for the condition looks like:

```
def Condition25_1763():
      # Condition terms
      # MEPMD5 25_1763 PROTECTED CONDITION CODE BLOCK
      if len(str(Location)) ==5:
            legal_zip=True
      else:
            legal_zip=False
      # MEPMD5 4a958480fd0245a0626fd58c2be1f277
```

→ *See MetaEdit+ Workbench User's Guide for details on using regeneration support.*

4 Conclusion

In this example, we have demonstrated a DSM for phone application development. With the domain-specific language we can model applications using phone concepts and execute them in a PC emulator or in the real phone device.

On the DSM definition side we focused on a few areas of language design: ports and importing external graphics to language notation. On the code generation side we described some options for relating manual code with the code generated from the models.

The modeling language is implemented as any other modeling language in MetaEdit+. It is completely open and thus it can be freely extended to cover additional requirements of modeling or code generation. You could for example modify the generator to produce native Symbian C++ code from the same models or extend the modeling language to cover a larger part of the phone framework than that provided by Python for S60. The choice is yours because with DSM you control both the language as well as the generators.