### CHAPTER 8

## MOBILE PHONE APPLICATIONS USING A PYTHON FRAMEWORK

In this chapter we describe Domain-Specific Modeling (DSM) for modeling and generating enterprise applications for mobile phones. The selected target environment is Symbian smart phones based on the S60 mobile phone platform (Nokia, 2005). The DSM solution was made to work with a Python framework, but to demonstrate that models in DSM can be independent of the generation target, we show how native Symbian applications in C++ were later generated too. The modeling language and thus the application design models are the same, only the generator and platform code are different. You may consider the Python framework as a subdomain of Symbian applications (Babin, 2005) as it can be used to make typical administrative applications but not, for example, graphical games or device drivers.

8.1 INTRODUCTION AND OBJECTIVES

Before discussing DSM we will briefly describe the Python framework, Python for S60 (Nokia, 2004). This framework provides a set of Application Programming Interfaces (APIs) to access the platform services and expects a specific programming model for the user interface (UI). At this point, we need to emphasize that in the world of mobile application development, UI is understood to cover logic and the whole application functionality, not just the look and feel. The modeling language described

Domain-Specific Modeling: Enabling Full Code Generation, Steven Kelly and Juha-Pekka Tolvanen Copyright # 2008 John Wiley & Sons, Inc.

160

### INTRODUCTION AND OBJECTIVES

here uses the widgets and services of the phone as its constructs, and follows the phone’s UI programming model. The generator produces Python code calling the phone’s platform services and executes the application in an emulator or in the actual phone device. Application execution requires that the Python for S60 package be installed, which includes the Python interpreter and S60 UI application framework adaptation connecting Python for S60 UI programming and accessing Symbian UI applications (such as phonebook, calendar, and camera). In this case, the phone platform and the Python framework were taken as given: there were no possibilities to modify them during DSM definition.

Target Environment and Platform The Python framework was developed to make application development easier when compared to traditional native S60 development based on C++. An easier entry is essential for opening mobile application development to a larger developer base, supporting possibilities for innovation, and lowering the barrier to entry for new developers. In addition to creating stand-alone S60 applications written in Python, Python for Series 60 also enables prototyping and concept development. The primary user target is Python programmers, but a separate development kit is also available for Perl (Hietaniemi, 2006) and a Mobile version of the Java Mobile Information Device Profile (MIDP) can be applied too. These SDKs thus target developers based on their preferred implementation language; with DSM, a single higher-level language can be used regardless of the underlying programming language, as discussed later in this chapter.

The Python for S60 release is offered as an extension to standard Python libraries (Rossum and Drake, 2005). It provides a subset of the underlying phone services and UI elements. For example, instead of over 10 different kinds of lists in native S60, the Python framework and its appuifw module provided only a few. In addition to supporting selected native Graphical User Interface (GUI) widgets from the S60 platform, the Python framework offers networking support for General Packet Radio Service (GPRS) and Bluetooth, Short Message Service (SMS) sending, and accessing other phone applications, such as camera or calendar. Python for S60 follows closely the S60 architecture and its UI programming model. The architecture of the Python for S60 environment is illustrated in Fig. 8.1 (Nokia, 2004).

The built-in appuifw module provides the key API. The Content\_handler object type facilitates interfacing to other UI applications and common high-level UI components. It is based on the notion that UI application interaction can be reduced to operations on Multipurpose Internal Mail Extensions (MIME)-typed content by designated handlers. Via the Canvas object type, general-purpose graphics toolkits can attach to the UI of a Python application. During creation of the DSM solution, Canvas support was not yet available.

Before going into the case, a few words about the application domain: Mobile applications must be built to fit the relatively small size of the phone display and the layout of a S60 application. Figure 8.2 shows the user interface layout and its relation to the services available in the appuifw API. Reading the UI layout figure from the top, the main application window may be set up to be occupied by a UI control/widget. A

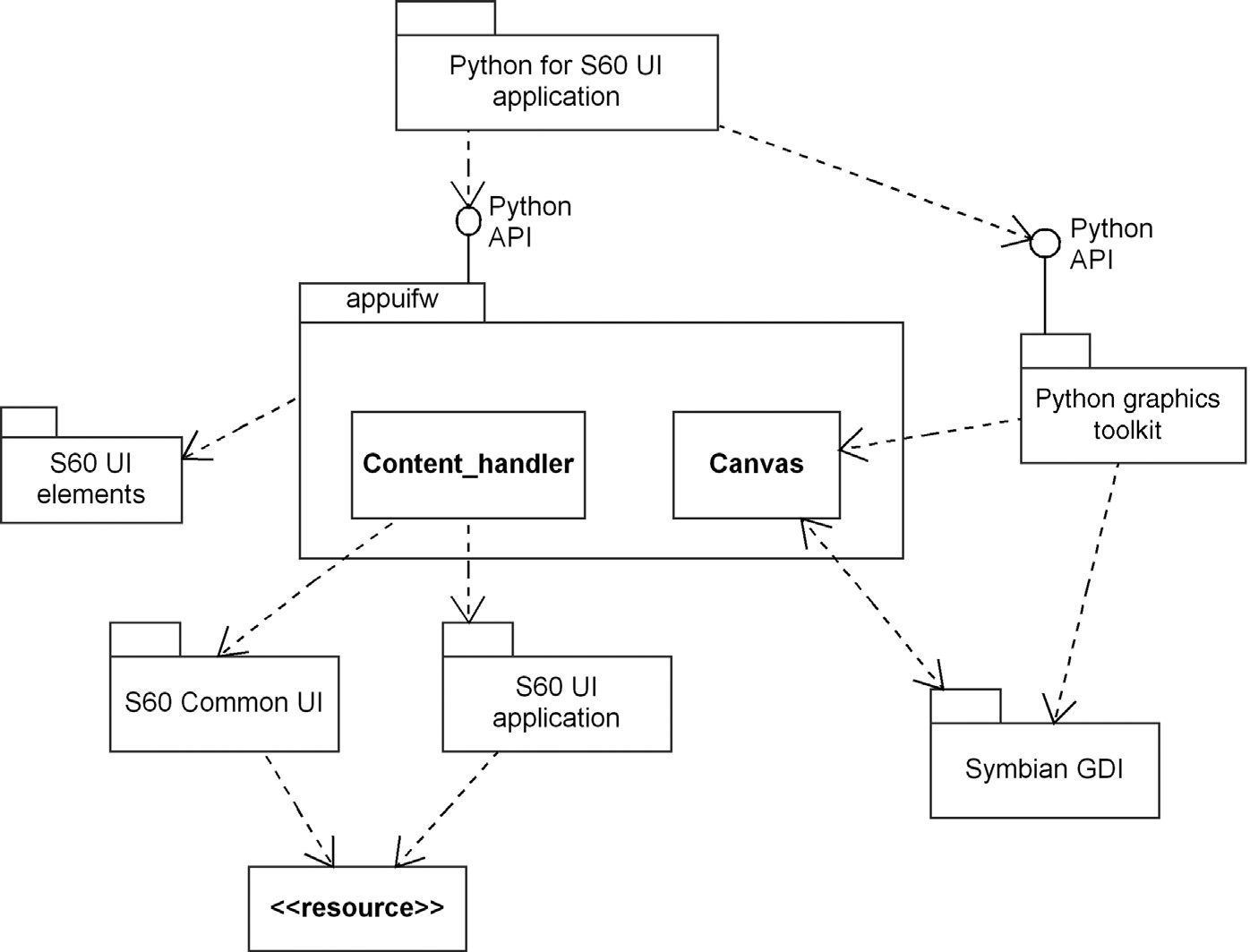


FIGURE 8.1 Architecture for application development with Python for S60

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Title ”appuifw.app.title” | | | | | |
| Navigation pane  ”appuifw.app.enable\_tabs(), acativate\_tab()” | | | | | |
| Main application window ”appuifw.app.body” | | | | | |
|  |  | | | |  |
|  | Dialog  ”appuifw.<dialog\_function>” | | |  |
|  | | |  |
|  | | | |
| Left soft key  ”appuifw.app.menu” | |  | Right soft key  ”appuifw.app.exit\_key\_handler” | | |

FIGURE 8.2 Python for S60 UI layout (Nokia, 2004)

### DEVELOPMENT PROCESS

multiview application can show its different views as tabs in the navigation pane and react as the user navigates between tabs—typically, by pressing the left or right selection buttons. Dialogs always take precedence over other UI controls and appear on top of them. The application uses two soft key buttons (left and right soft keys) to navigate in the application. Usually, the left soft key action in the dialogs is to accept, and the right soft key to cancel or navigate back.

DSM Objectives The objective of the DSM solution was to make application development possible for a fundamentally larger audience: people having little or no experience in programming Symbian/S60 applications. Following the original idea behind the Python framework of making application development easier, the DSM approach was seen to make it even easier while still guaranteeing that the framework is used correctly. The main idea was to get a situation where the developer could draw a picture of the application and then generate it to be executed in a target device. This quick and easy development approach was considered important for companies who wished to make some of their in-house applications mobile but did not have enough experience in mobile application development. The language would have the rules of the architecture and UI programming model so that a developer could focus on application functions and design, not on their implementation details.

8.2 DEVELOPMENT PROCESS

DSM definition started when the first public release of the Python framework was made available as a technology preview. During the DSM definition process, the framework’s API became larger and about 15% of the APIs changed before the official release (version 1.0, Nokia, 2004). Since then new services have been added to the API but the basic structure and programming model have remained the same.

The definition process followed a prototyping approach, first making an example as a proof-of-concept and then gradually extending the language to support the whole Python framework. The first prototype was made by choosing a sample application to be modeled and generated. The sample application was the same as used in the documentation of the technology preview, so its design and implementation code was available. This application supported car pool users and used only two widgets, one asking the user to enter travel information and the other informing the user on the application state via a note dialog. On the application service side, the application used SMS to send text messages via the operator’s network.

Development of the prototype took 1 day, during which the language and generator were directly defined into a metamodeling tool and the sample application was modeled and generated. At this stage, the modeling language had only three objects and a single relationship to connect them. The generator followed a simple flow navigation (similar to the Voice menu application in Chapter 7) to produce the application code. This code was structured similarly to the manually written application code made available as an example in the documentation of the Python for S60 technology preview. The DSM implementation did not really cover application navigation or canceling—those were also lacking from the technology preview example.

The actual DSM implementation started once the framework became available in the first public release (version 1.0), in which about 70% of the framework APIs were the same as the version used to make the prototype DSM solution. The public release included several sample applications and those were taken as reference implementations, guiding the language and generator definition. Implementation of the DSM solution described in this chapter took about 10 man-days and was done by one person not involved earlier with S60 or Python. The implementation started from the metamodel of the prototype language and was done directly in a customizable metamodel-based tool. The review and testing of the DSM were supported by S60 and Python experts. Two experts were available to review the language definition and one expert tested the language within the created DSM tool. Testing with the tool involved using the language to model sample applications and running the generator to execute the applications. The availability of experts helped most in the beginning to get the basic structure of the DSM language defined.

Two review and testing phases were carried out during the process. The first testing was performed after having defined over half of the language, so its basic structurewas clear. The second testing was done when the language was already defined, something you may consider a beta version. Most of the feedback and comments were related to code generation: finding a consensus among the four programmers involved on a good structure for the Python code to be generated. Finding good structures for the code was not just for the sake of DSM as that was needed anyway. Most testing work was done by the language definer by modeling about 10 small to midsize mobile applications using various constructs of the language. The language for Python for S60 was relatively easy to test as the domain could be examined by running the applications within the mobile phone, either in the real target device or in a PC-based phone emulator.

8.3 LANGUAGE FOR APPLICATION MODELING

The objective of the language was to make application development as easy and natural as possible. The approach to achieve this was by using modeling concepts directly based on the phone’s services and UI widgets and by following the already used “cartoon-style” of UI application specifications. Thus, domain concepts such as “Sending text message,” “Note,” “Form,” “Pop-up,” “Browsing web” became candidate concepts for the language.

The application logic follows the flow of navigation, mainly using UI layout and connections between UI widgets. This cartoon-style UI specification, proceeding from one dialog to another, was also used in many companies developing mobile applications. In it, an application is specified using screenshots of relevant user interfaces along with a textual description of the stage and its relation to other stages.

In some cases, mock-up applications and “phones” made from cardboard are used, with application logic shown by swapping in different screens drawn on paper. Simply put, with DSM we aimed to replace these paper prototypes with a design that can be executed.

8.3.1 Modeling Concepts

The modeling language consisted of three main kinds of elements: dialogs, main UI controls, and phone services. We started with dialogs as they were the most stable and had clear characteristics. They were already working well in the first concept demo. After the rest of the dialogs were defined, the user navigation and application flow could be addressed with the language. The language concepts were named by the customer. Most of the names for language concepts came directly from the native S60 platform and from the Python for S60 concepts.

Basic Dialog Concepts The modeling concepts for dialogs included the following:

. Note: Displays a note dialog with text and a notification icon that indicates information, error, or confirmation. Later, Note was extended to allow writing custom code in the model element. In addition to writing plain code, reusing functions written in Python was also supported. For this purpose, the Note concept had an optional Function property type that referred to a Function in the Function library. The available functions could thus be imported as code files and refered to in models.

. Query: Displays a query dialog with a single field. In the language, this concept had a property Prompt to enter the query label and Type to choose what kind of value the query expects. This latter property was implemented as a list of fixed values to be chosen from: “text,” “code,” “number,” “date,” “time,” or “boolean”. The query dialog can include an optional initial value and the value entered is stored as an optional, but unique, return variable. If a return variable is not given, the value is treated as a local variable. This structure was also used in other concepts, allowing values to be defined that are used globally inside the program. All queries return None if the user cancels the dialog. Further details on navigation will be discussed later.

. Multiquery: A two-field text input dialog. Both queries show a prompt label in the same dialog and have separate return values.

. List: A dialog that allows the user to select a list item and returns the index of the chosen item or None if the selection is canceled by the user. List items are Unicode strings.

. Pop-up menu: A dialog list representing the menu contents with a label and a list of Unicode strings. Similar to the list, it returns the index of the chosen item and an optional return variable was included for the pop-up menu too. On cancellation, None is returned.

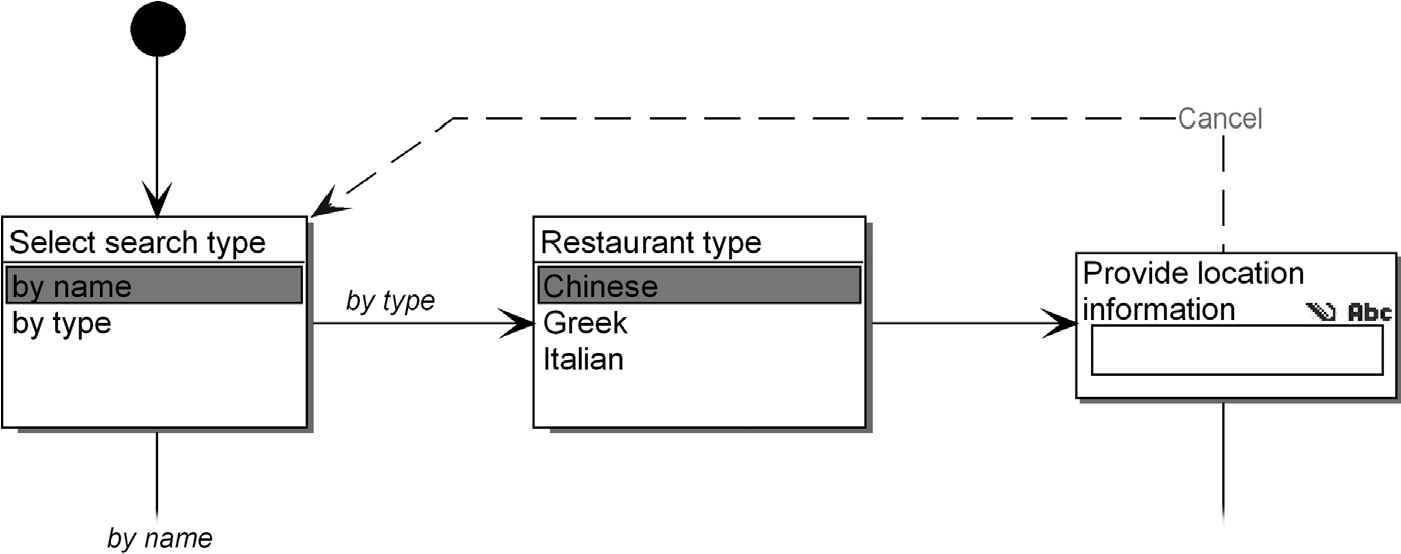


FIGURE 8.3 Sample model illustrating navigation between dialogs (two pop-up menus and a query)

Specifying Navigation with Relationships To describe application logic and navigation flow, dialogs—and later main controls and accessing phone services—could be connected with directed relationships. Figure 8.3 illustrates a sample model where dialogs are connected via directed relationships. Hence, there is no need to explicitly specify separate soft key buttons (two context sensitive buttons, see Fig. 8.2) as modeling properties for each dialog type. Having an extra place to define soft keys would just add extra modeling work, as navigation flow needs to be defined anyway. The directed relationship concept in the language could have a property type to describe the name of the soft key button, but as they are predefined in S60 and follow standardized UI conventions, their naming was not needed. Naming depends on the UI element, but typically, soft key buttons are named OK and Cancel. Symbian C++ development (Babin, 2005) sets different requirements for localization from the Python framework discussed here. In native C++ development, a typical approach is to add a localization entry as an identifier or generate it, if possible, for those concepts that are mapped to localization resources. During code generation, resource files can then be generated and localization data for different languages can be added.

Describing Choices with the Language Dialogs that provided multiple choices needed a Choice construct to enter the selection made for the specified connection. In Fig. 8.3, this situation is illustrated with the selection of the search type. Choosing to search by name opens a pop-up menu to choose the restaurant type. To be precise, the Choice property was added to the start role of the relationship to be close to the dialog element where the choice is made.

In the metamodel, the list items property and choice property used the same property type so that existing list values could be selected as choices, instead of writing them manually into the design models again. If no value was given for a flow, then that flow was the default for all list items without their own flow. This allowed minimizing the modeling work. For example, in Fig. 8.3, it is enough to draw only one flow from the restaurant type selection to the query, instead of modeling all three possible choices from the pop-up menu separately.

Canceling (pressing the right soft key) was also supported with a separate directed relationship type. Its use, however, was not defined as mandatory as in most design situations it is not necessary to specify. Normal canceling behavior could be derived by following the application flow backward and only abnormal cases needed to be specified. This “convention not configuration” approach worked well here and greatly minimized the modeling work—in most cases, only forward navigation needed to be specified.

Relating Manual Code to Models Early on the customer raised the option of manually entering Python code in designs. Perhaps this was a way to allow a back door for manual programming, but the sample applications in the Python framework did not indicate any need for it. However, to satisfy the customer and show that the old approach can be applied as well, both default navigation and canceling were extended with the possibility to add custom code. In the metamodel, a code property was added. A more sophisticated approach was to refer to functions available in a library. This was defined in the metamodel as a Function property similar to the Note object. The Function included a whole Function object with its own properties, such as name, parameters, code, and documentation properties. Python code written in relationships was used during code generation, and editing it in generated code was also made possible with protected regions. See later in this section for details.

Modeling Concepts to Access Phone Services Mobile applications usually need to access phone services. For this purpose, the modeling language was extended with additional concepts, including the following:

. SendSMS concept: A given string starting with a keyword, followed by message elements, is sent to a recipient number. To include variable values in the string a Message elements property was defined in the metamodel to include any of the Return values saved from dialogs or from the main controls. Here, the language integrates the use of a Returnvalue as a Message element. If the name of a returnvariable changes in the application design, there is no need to manually propagate the change elsewhere in the models. A message element could also be a user-defined delimiter. Common delimiters, such as comma and dash, are provided for selection in a predefined list property. If a message element is taken from a complex data type (e.g., from a Form including several fields), the chosen value is referred to by its index.

. Open: For accessing files and browsing the web, an Open concept is used. This concept has a property, a target address, or a file name. In later versions of the Python framework, a stand-alone opening option was added to enable opening in a separate processes. For example, a browser could be started in a new process outside the current Python application process. Because this standalone method of opening did not have the possibility of canceling the process inside Python, it was defined as a new language element instead of simply adding a Boolean property to differentiate the two access methods. This made it easy and clear for the metamodel to forbid adding a cancel flow to a stand-alone open object.

. The Start exe concept was defined to access native prebuilt phone applications, such as Calculator, Calendar, Camera, and so on, from the Python application. This modeling concept, Start exe, had a list property of available S60 phone applications. The first release of the Python framework did not provide access to a phonebook or making calls, but support for them could later be implemented by simply extending the current concept or defining subtypes for it (e.g., for making a phone call with the possibility of entering the phone number directly or accessing the phonebook first to choose the number). Entering the number to be called could already be supported with the Query concept, and then passing it to the phone call module could send the text message.

Specifying Domain Concepts with Internal Behavior In addition to dialogs and services, a Python for S60 application also usually contains some more complex user interface controls. The main UI controls are different from previously defined dialogs as they fill the whole display (see Fig. 8.2) and have a richer internal structure with their own state behavior. Their structure in the language definition is hence richer too. It is usually best to specify such complex internal behavior with a new language, as in the case of the watch in Chapter 9. Here, UI controls had rather limited internal structure that could be handled with menu connections and property types that describe behavior. Therefore, an additional language for the UI controls was not needed, although it could easily be added if needed, with subdiagrams for each UI control.

On the menu side, all UI controls could change the menus provided via the soft key buttons. The soft key buttons and menus for UI controls are managed by the underlying S60 platform, but the menu items available with the left soft key could be extended depending on the definition given for the UI control. Defining menus with the language was made possible via a specific flow, in which the flow end (a role called Menu item) allowed specifying a menu label and a target element. This target was usually a UI element or a phone service. The use of multiple relationships for each menu item did not allow specifying the order of the menu items. Ordering was not considered important as the Python framework itself did not provide full control of menu item ordering but only allowed adding new items. Item ordering could have been supported by adding order numbers for the flows or by using separate menu item elements, in other words, using a structure similar to the one for List.

Other internal behavior was dependent on each UI control and thus was captured as part of the respective type. The main controls include the following:

. Form: A dynamically configurable, editable multifield dialog. Form is characterized by fields, which can be either a simple input entry field or a Combo field. In the language definition, Field was defined as a collection of Form fields and Combo fields. A Form field element is defined by a label, a data type, and an initial value, and a Combo field is defined by a label and a collection of values. Because Form can be used either for editing the field values or just for viewing them, a property to choose between Edit and View modes was added to the Form element. This single choice allowed automating the Options menu content and the behavior of the Form without any additional definitions in the model: If the list is editable, the Options menu includes the save menu item and closing the form after edit asks whether the changes made should be saved. Boolean properties are also defined for Form to disable the automatic support allowing the application user to edit the labels in the form fields, and to specify a double-spaced layout (one field takes two lines as the label, and the value field is on different lines). To save the form, a return variable, similar to the dialogs, was added as a property of the Form concept. . An editor UI control: A Text Editor modeling element with a title property, an initial text property to show predefined text when starting the editor, and a property for entering editing code. Similar to the other UI controls, a return variable is used to save the given text for later use.

. Listbox: Listbox in a mobile application shows a list of strings or a list of tuples of strings. The Listbox modeling concept has a property to enter a Listbox name that is visible on the screen and a content creation function and its return variable. The content creation function could be entered directly as text into the model or chosen from a list of available library functions. More of these options will be discussed next as well as in the sections on generator definition.

As mentioned, the main UI controls raised the need to specify some Python code related to models. Three options were provided:

. Code could be added directly to selected model elements. This was considered best for cases where just a few lines of code may be needed, like an optional save validation function, which checks the form content before it is saved. . Code could be reused from the library. Here, a model refers to code available as a function, and the model only includes the parameters to be passed to the function. The modeling language was made to allow both black-box and white-box component libraries so that the modeler could also write function code. To support code reuse from a library, the modeling language was extended with a library concept added to the UI controls, navigation flows, and Note dialog. If this property has a value, it is considered to refer to a function. During code generation, the functions used are included in the generated code.

. Code could be added to the generated code. For this purpose, protected regions were defined in the generated code. Basically, all previously defined properties were also marked as protected regions so that they could be changed after generation. This also required that empty blocks be generated in case the developer wanted to add code only after generation.

Other Notable Language Constructs Later, the language specification was extended by customer request to allow an optional naming policy. This was suggested to allow modelers to give human-readable function names that are visible in the generated Python code. The function name property was added to the modeling concepts and its use was made optional and left to the modeler’s discretion; if not given in the model, unique function names are generated, minimizing the modeling work.

The specification of navigation flow required ways to specify application start and end. For this purpose, Start and Stop concepts similar to state machines were added to the language. As not all applications are necessarily closed during user navigation, the Stop element was extended to choose a policy for closing the application. Application exit was set as the default choice but the Stop element allowed choosing a wait policy. The wait policy is typical for applications that have several views and ending the navigation in one view should not close the whole application but allow for choices in other views or external processes that have been called, for example, via the Start exe concept.

All possible Python framework services, such as retrieving phone location information from the network, accessing drives of the phone, or copying files, were not included in the language as their own concepts. Their use could be supported with the previously described function calling capability: they could be defined as ready library services that are used in the designs with the Library concept.

Finally, a separate comment element was added to the language to allow attaching free form textual descriptions to models. This concept was extended with model checking capabilities: its representation in the design showed the results of a selected model checking report. Figure 8.4 shows a metamodel for the modeling language showing the legal connections between different object types of the language.

8.3.2 Modeling Rules

The basic metamodel included language constructs and their connections, but these did not cover the domain rules and the mobile UI programming model. Extending the language with rules would prevent errors and enable design consistency at modeling time. These were considered important as most application developers, similar to those using plain Python for S60, don’t have much experience of mobile application development.

Rules Checked During Modeling The behavioral part of the application was largely modeled using directed relationships for specifying user navigation, view

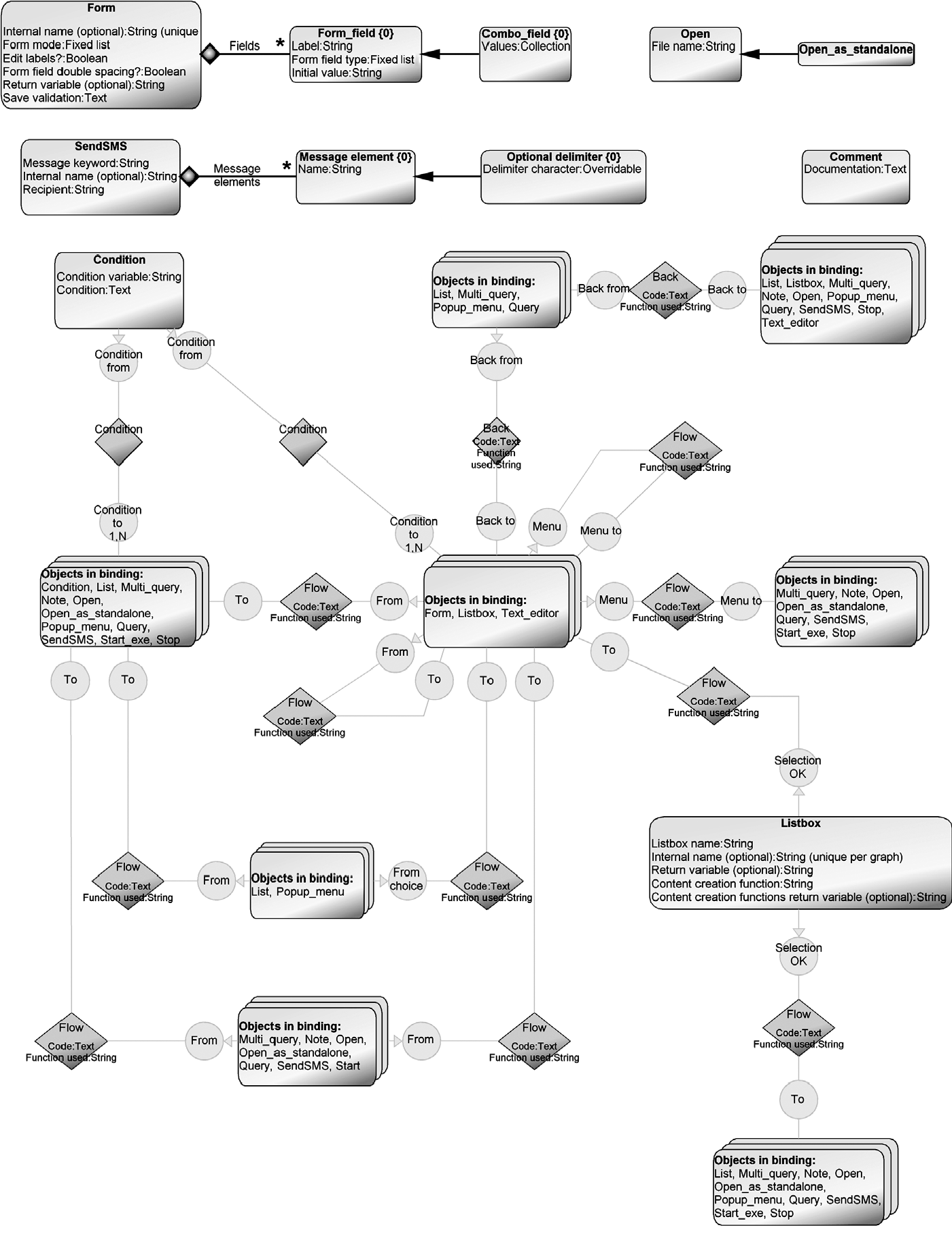


FIGURE 8.4 Metamodel of the language

structure, and access to the phone services. Here, general rules from state machines were reused and extended:

. Only one event can be triggered from the start state and canceling the start is not allowed. If canceling is defined in the application design (i.e., Cancel is pressed

in a dialog after a start stage) the generator produces an application exit code (see Section 8.5.2 for details).

. There can exist only one start object per application design, but if the application consists of multiple views, each view can have its own start object.

. The optional function name was declared unique to enable unique Python function naming during code generation.

. The return variable used in most modeling elements was made unique to support variable naming and to allow reuse of the entered values within the application.

Rules that dealt with the possible navigation actions, such as accept, opening a menu, and cancel, were defined on the connections between the modeling concepts. Some of the rules are as follows:

. For some language elements, normal navigation flow was defined to allow only one target element. For example, in S60, after showing a notifier for a couple of seconds, only one UI element or phone service can be triggered. Accordingly, the metamodel included a rule that allows only one flow from the Note element.

. The exception to the above navigation rule was allowing multiple targets for List and Pop-up dialogs and for accessing menus from UI controls. . A particularly good choice for keeping models simple and minimizing the modeling work was to base the language on convention rather than configuration. For example, navigation by pressing the cancel button did not usually need to specified at all.

. Specifying optional cancel navigation flow was restricted to a few UI elements only. List, Multi\_query, Pop-up, and Query were defined to be possible sources for cancel (with the Back relationship, see metamodel in Fig. 8.4). Only one cancel per model instance was allowed.

. As the main UI controls (Form, Listbox, TextEditor) did not have a cancel option, their navigation was based on closing the control or choosing from its menu. The latter, closing the application from its menu, was specified in the language by allowing menu items to also include a Stop object.

Rules that dealt with accessing phone services include the following:

. Every phone service allowed triggering one UI element or other phone service. . Creating a new thread with Open as stand-alone was modeled similarly to the others, but canceling it was not possible from the program calling the new thread. This was also the case with starting external applications outside Python—their dialogs and other widgets could not be handled or synchronized with the modeled Python for S60 application.

Guidance Rules Some of the domain rules were defined to be checked only by user choice. For this purpose, model checking reports were made, to be run when needed or alternatively to be made available during modeling by showing a comment element in the model. In other words, the comment element shows the result of running a model-checking generator that reports on possible errors and gives guidance if it detects that the model is incomplete. The latter was considered useful to guide developers in the beginning as it showed some of the tasks needed to complete parts of the design. The reports checked the following:

. If navigation flow was interrupted so that the application would stop before exiting.

. If choices entered in the flow from Condition, List, or Pop-up had more than one empty choice value or several identical values. . If menu items started navigation flows that started more than one action. Longer chains were considered to lead to navigation paths that are too difficult to understand by application users. Placing this rule in the guidance report still allows one to make such designs (and thus applications also) so the rule was more of a suggestion than an absolute rule.

8.3.3 Modeling Notation

The notation was created in parallel with the metamodel. As the modeling concepts were largely taken directly from the UI elements, it was also natural to take the symbols from the actual phone UI. This created a powerful semantic mapping from the model to the running application. Figure 8.5 illustrates symbol definitions for the Form, Query, and SMS sending concepts. Form and Query are identical to their appearance in an executable application. Form includes its own compartment for the Options menu, from which users could draw connections to add menu items for the connected application elements. Query shows a prompt and an optional initial value as

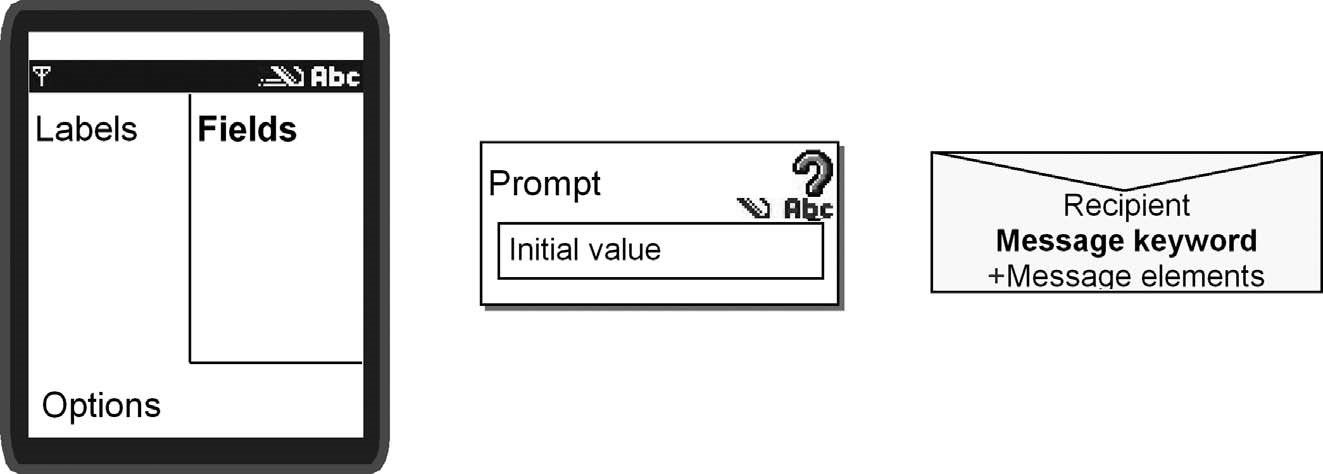


FIGURE 8.5 Defining the notational elements for language concepts Form, Query, and SendSMS

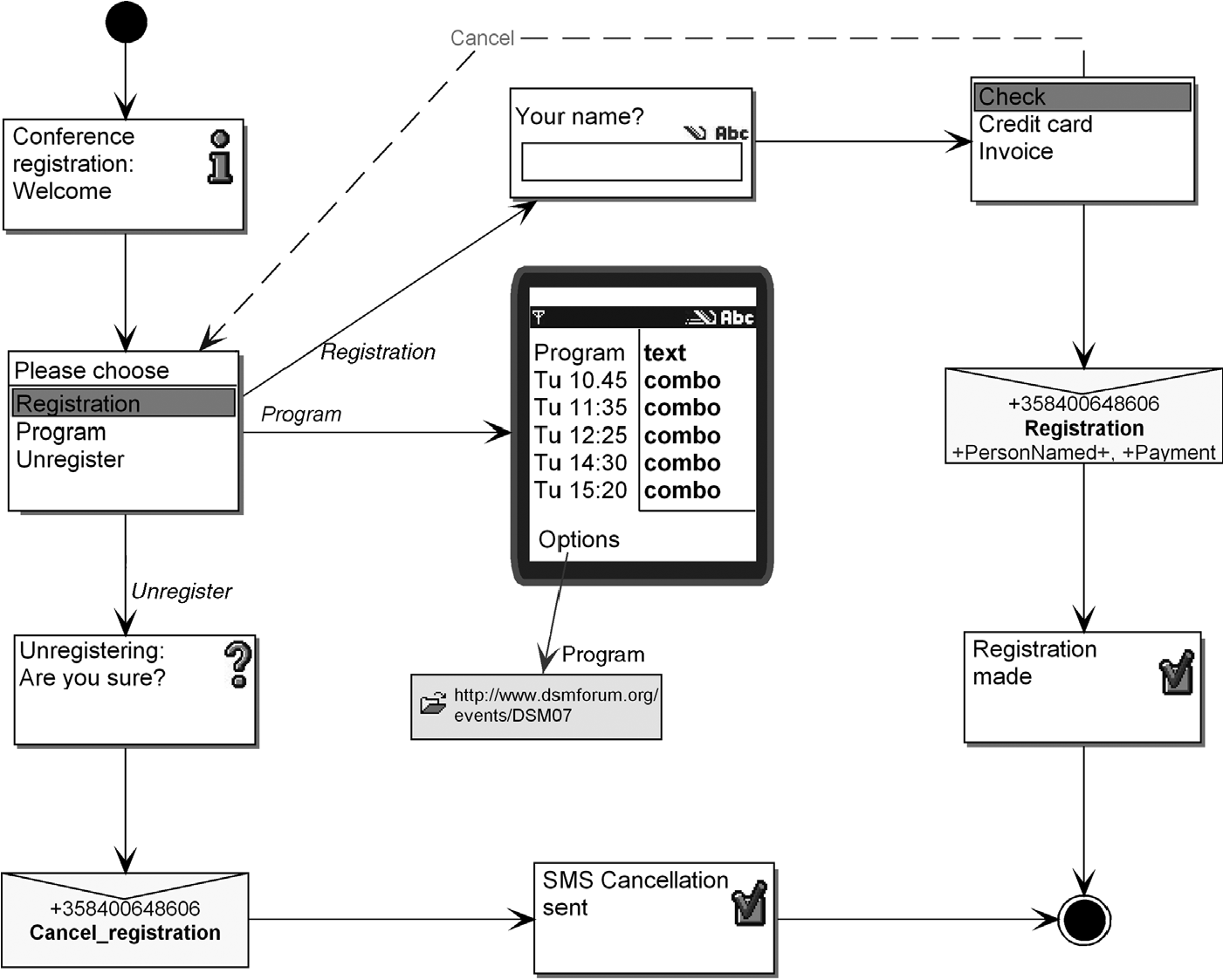


FIGURE 8.6 Design of a conference registration application using the DSM language for Symbian/S60 mobile phones

well as an icon for text entry, number entry, or Boolean entry, as specified by the Query type. Text message sending was shown by an envelope symbol showing the complete message content and recipient number.

Other modeling concepts were defined in a similar way and their appearance is illustrated in the example model (Fig. 8.6). Navigation was illustrated using directed arrows: normal navigation with a solid line with an arrow and canceling with a dotted line and “Cancel” text in the relationship. Menu items were also specified via relationships, but those were connected to the bottom left port compartment in the main UI control. For example, Fig. 8.5 shows the Options menu port compartment that allowed specifying new menu items.

8.4 MODELING PHONE APPLICATIONS

The modeling language was tested immediately after adding to or changing the metamodel. This ensured first-hand experience of using the language.

### MODELING PHONE APPLICATIONS

8.4.1 Example Models

The DSM language is illustrated in Fig. 8.6 with a sample application design. If you are familiar with phone applications, like a phone book or calendar, you will probably understand what this application does. A user can register for a conference using text messages, choose a payment method, view the program and speaker data, browse the conference program on the web, or cancel the registration. As can be seen from the model, all the implementation concepts are hidden (and need not even be known by the modeler).

8.4.2 Use Scenarios

The basic use scenario for building an application approaches the ideal of drawing UI elements and phone services on a canvas, connecting them and then running the generator to produce the application and run it in the target device or an emulator. Figure 8.7 shows a sample application design.

As the application size was considerably smaller than in native C++ development, localization, reuse, and support for product lines were not considered relevant. The application size was usually just one diagram, or possibly many small ones for a multiview application. Applications were usually less than 300 lines of Python code. The application modeled in Fig. 8.6 is 145 lines. Reuse is not considered in the language other than allowing sharing the same function library. At the model level, the

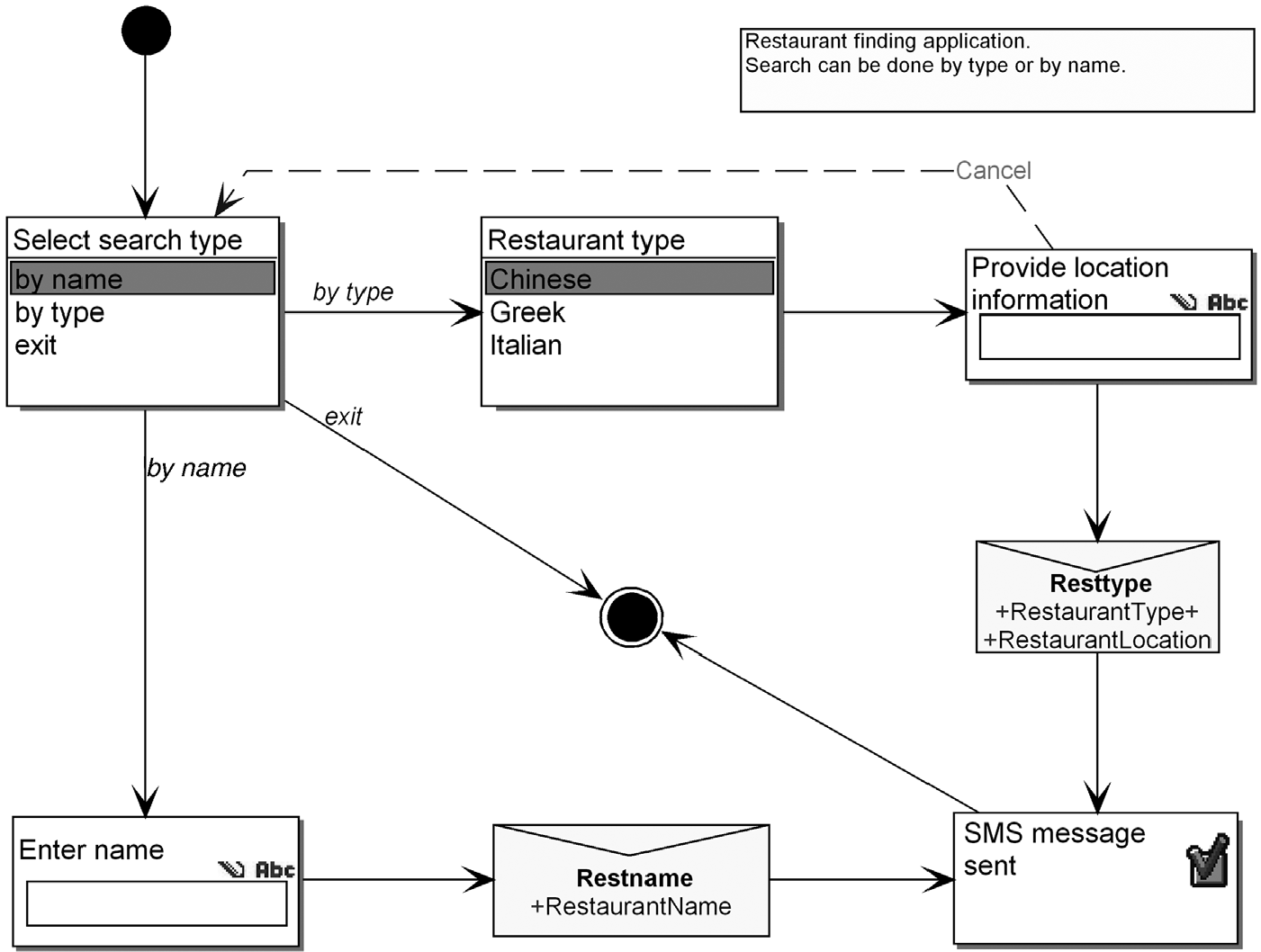


FIGURE 8.7 Sample application design

modeling tool used does however allow one to copy or share the same design elements between different applications, though. Typically, one developer makes the whole application and there is no need to support multiple views of the design or support multiple simultaneous modelers.

8.5 GENERATOR FOR PYTHON

After some example designs, code generation was targeted. The prototype DSM language, built as a proof-of-concept, applied a simple tree structure where the generator navigated from the first dialog to the last one and produced code for them that called the Python framework. Although this approach allowed generating applications similar to the example applications provided with the Python framework, it was very limited. It expected designs and application navigation structure to form a tree and did not allow for variable handling or differentiate local and global variables. As the manually programmed sample applications provided with the Python for S60 package used function definitions, calling them via a dispatcher, the customer thought this would be powerful enough for code produced by a code generator too. The other alternative, use of state machine based approaches (as in Chapter 9), was considered over kill.

From the designs expressed with DSM, the generator produces code that can be executed both in an emulator for testing purposes and in the target device. Next, we describe the structure of the code generator and then show samples of the generated code.

8.5.1 Generator Structure

Following the chosen function-based approach, each UI control, dialog, and phone service was implemented as a corresponding Python function definition. The main UI controls, like Form, also had internal functions. This approach also influenced the generator structure: each modeling concept mapped to one or more generator modules; one generator module takes care of List, another SMS, and so on. The general structure of the generator is illustrated in Fig. 8.8. An autobuild generator produces the Python script and starts an emulator to execute the application.

The generator first produces the imports necessary for the application (like appuifw for UI elements or messaging for sending text messages) and defines the functions used from the library. Then the main part of the generated code is taken from dialogs, UI controls, and phone services described in the design models. These generator modules, identifiable by their language concept name, are collected together in a gray rectangle in Fig. 8.8.

As many of these modeling constructs have some shared behavior, such as naming and navigation, common parts were implemented in shared generator modules. Generator modules “\_Internal name” and “\_Return variable name” are widely used during Autobuild and thus their use in the call order is not shown in Fig. 8.8 to make the figure more readable. When it comes to the common code, one generator module (Python script) created a part common to all Python for S60 applications: the

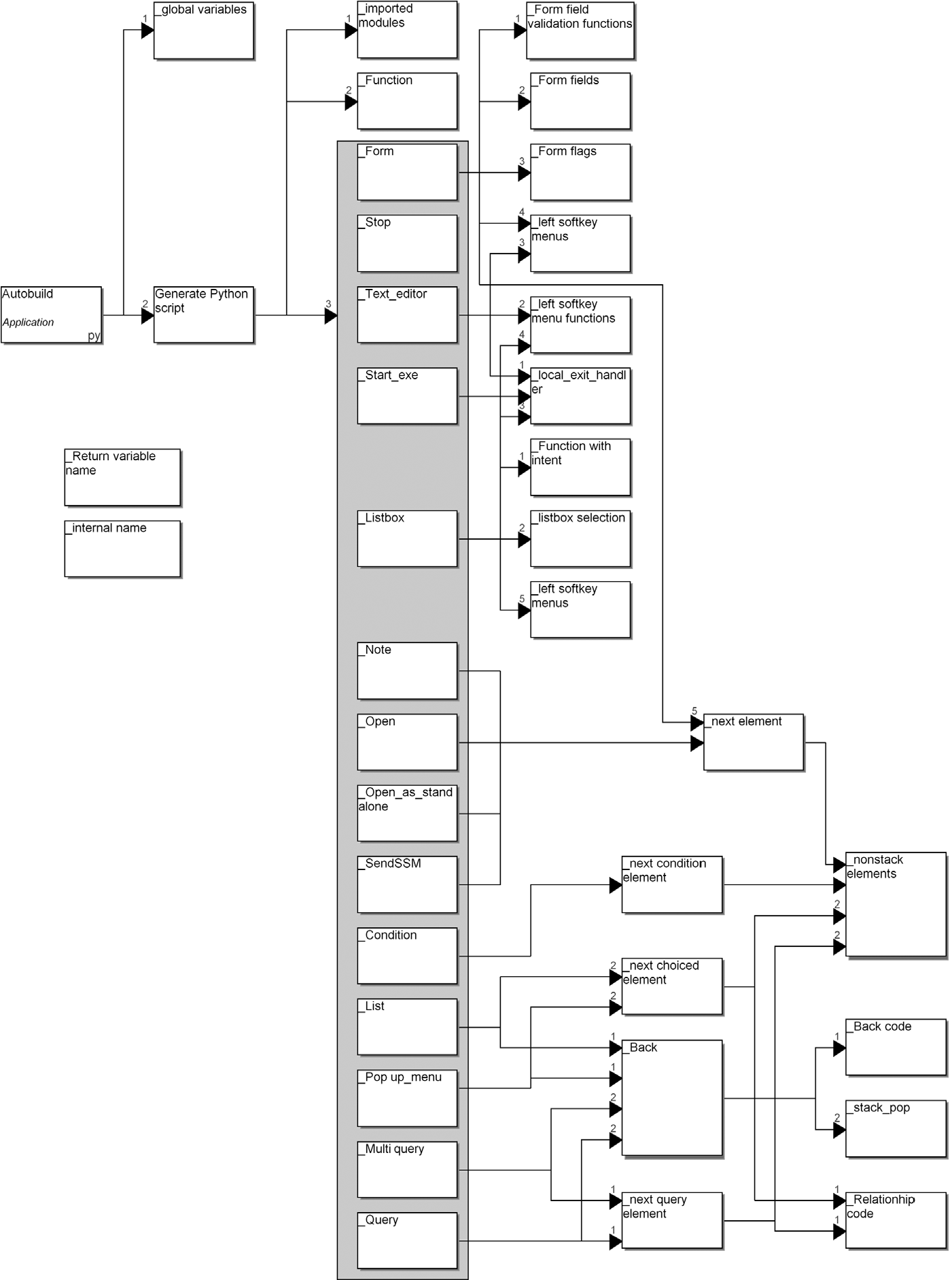


FIGURE 8.8 Structure of the Python for S60 generator

navigation stack handling and the main dispatcher to start the application. This code was nearly identical for every application and thus that generator module did not use much data from the design models. The results of that module are shown in the sample application code in the next section: see lines 128–145 in Listing 8.5.

A Basic Generator Module for One Dialog Concept Let’s look in detail at the generator definition for one of the dialog types: pop-up. It opens a dialog with a prompt label and list of choices. Based on the choice made, the next function is called. The generator definition is described in Listing 8.1. It is defined in the generator definition language used by the MetaEdit+ tool. Underlining in the listing refers to the language concepts: It shows which parts of the code are taken from models and thus what can vary between the applications.

Lines 1 and 17 are simply the structure for a generator and line 2 is a comment for the generator module. Line 3 creates the function definition signature and line 4 is a comment for the Python code. Function naming in line 3 calls another generator module that creates a unique function name if the developer has not given a name for the pop-up menu in the application design model.

Listing 8.1 Code generator for pop-up menu.

1. Report '\_Popup\_menu'
2. /\* Produces popup menu-style dialog code \*/ 03 'def ' subreport '\_Internal name' run '():' newline
3. '# Popup menu ' :Label newline
4. if :Return variable (optional); then 06 ' global ' subreport '\_Return variable name' run newline
5. endif
6. ' choices' oid ' = ['
7. dowhile :List { 'u"' id '", ' }
8. '"]' newline
9. ' index = appuifw.popup\_menu(choices' oid ', u"' :Label '")'

newline 12 ' if index <> None:' newline

1. ' ' subreport '\_Return variable name' run '= choices' oid

'[index]' newline

1. ' if index == None:' newline
2. subreport '\_Back' run 16 subreport '\_next choiced element' run

17 endreport

Lines 5–7 declare the return variable, if this pop-up has one. Lines 8–10 create a temporary variable that includes the choices given for the list in a Unicode string. Line 11 produces the actual pop-up call using the service from the Python framework. It gives as parameters the choice elements defined for the list and label of the pop-up. If no label is given, an empty Unicode string is generated. Lines 12–13 save the selection made in the pop-up into a global variable (done in a subreport “\_Return variable name”). If no return variable name is given, then the choice value given by the application user will not be used other than for choosing the navigation path.

Reusable Generator Modules: Navigation As several concepts require similar code to be generated, such as navigation flow and function naming, parts of the generator definitions are made into modules called by other generator modules. For example, the generator module “\_Back” is used by several dialogs to generate canceling code. The generator module “\_next choiced element” in line 16 is also used by List to generate transition code based on the choices made in the list dialog.

Part of this generator module is illustrated in Listing 8.2 below. To allow drawing models where choices don’t need to be explicitly specified but can be left undefined, the models are read twice: first, for cases where all list values are specified (like in the pop-up for choosing an action in Fig. 8.6) and then for cases when a choice does not guide navigation but is used for getting user input for later use (like the list for selecting a payment method in Fig. 8.6).

Listing 8.2 Code generator for \_next choiced element.

1. Report '\_next choiced element'
2. /\* Reports next choiced element for lists and popups \*/
3. do ~From choice {
4. /\* If there is specified value in Choice property \*/
5. if :Choice <>'' then
6. ' elif choices' oid;1 '[index] == u"' :Choice '":'

newline ...

1. /\* If Choice property is not specified \*/
2. if :Choice ='' then
3. do >Flow~To.() {
4. ' else:' newline
5. ' return (' subreport '\_Internal name' run', '
6. subreport '\_non-stack elements' run')' newline
7. }
8. endif
9. }
10. endreport

Line 3 follows the navigation flow from Pop-up (or List) based on the choice roles leaving it. If a Choice value is specified on the navigation arrow, the generator produces code for the selected choice (lines 5 and 6). Line 23 analyzes only cases where no choice is specified on the arrow: it creates a single choice with the Python else construct. It should be noted that multiple undefined choices or several identical choices are already checked from models: an application developer is informed about Pop-up and List elements, that have several choices with empty or identical values. Although such checking could be performed at code generation time, it was considered much better to inform the developer as early as possible about errors and conflicts.

Generator Module for Accessing Phone Services On the phone service side, Listing 8.3 describes a code generator definition for producing code for sending text messages. After creating a function definition and a comment (lines 3–4), message elements sent (which can also include delimiters) are declared as globals, so they can be referred to from inside the text message sending function (lines 5–9). As the metamodel is made so that message elements are actually return variables given in design models, this allows access to data given elsewhere in the application. Lines 10– 19 create the message content as a string that is then used as a parameter when calling the Python for S60 API in line 21. Finally, line 22 calls a commonly used generator module to create a navigation call to the next function.

Listing 8.3 Code generator for sending text messages.

1. Report '\_SendSMS'
2. /\* Produce SMS sending code \*/
3. 'def ' subreport '\_Internal name' run '():' newline
4. '# Sending SMS ' :Message keyword ; newline
5. do :Message elements {
6. if type ='Message element' then
7. ' global ' id newline
8. endif
9. }
10. ' string = u"' :Message keyword; ' "'
11. dowhile :Message elements {
12. if type ='Optional delimeter' then
13. '+"' id ' "'
14. endif
15. if type ='Message element' then
16. '\' newline ' +unicode(str(' id'))'
17. endif
18. }
19. newline
20. ' messaging.sms\_send("' :Recipient '", string)' newline
21. subreport '\_next element' run
22. endreport

As can be seen for the generator definition, Python also sets layout requirements for the code generation definition: indentation is used to express nesting. This Python feature made generator definition a little more tedious but also required creating nearduplicate generator modules. For example, the only generator difference between a normal function definition and the Listbox content creation function is that the latter is indented to be part of the Listbox function.

Different Generator Usages This basic generator produced plain code to be immediately viewed. For alternative code generation tasks, three additional generators were defined. They all called the basic generator for the actual code generation and just guided the use of the generation result for alternative needs: One generator saved the application into a file using the application name as the file name. A second generator uploaded the application into the emulator to be executed via the Python interpreter. The third generator added the application to the main grid of the phone (like a desktop), similar to other main phone applications. This generator also included information about the application number and its place in the grid in the generated code. It would also be possible to have a generator that creates a stand-alone application installer for sharing the applications, by calling the build tool for application packaging.

8.5.2 Generator in Action

Application designs were executed in an emulator and in a real target device by running the generator. This was part of testing the DSM solution. In addition to executing the generated applications, model checking reports were also tested to show missing or incomplete parts in the design. Listings 8.4 and 8.5 illustrate code generated from the model shown in Fig. 8.6. As specified by the generator, code is structured into functions, one for each dialog and service. In the listings, underlining is used to mark those parts of the code that are taken from models. To make code more readable, the listing does not show protected regions. After all, they were not needed in this sample application.

Listing 8.4 Python code generated from Fig. 8.6.

1. import appuifw
2. import messaging

03

04 # This application provides conference registration by SMS.

05

#Globals

06

Setdata

=

""

07

Payment

=

""

08

PersonNamed

=

""

09

1. exit\_flag = False
2. call\_stack = []
3. lock = e32.Ao\_lock() ...
4. def List3\_5396():
5. # List Check Credit card Invoice
6. global Payment
7. choices3\_5396 = [u"Check", u"Credit card", u"Invoice"]
8. index = appuifw.selection\_list(choices3\_5396) 46 if index <> None:
9. Payment = choices3\_5396[index]
10. if index == None:
11. return (Choice\_list, True) 50 else:

51 return (SendSMS3\_677, True)

52

54

appuifw.note(u"Conference registration: Welcom

e", 'info

')

55

return (Choice\_list

, False)

65

def Open

3\_1803

():

53 def Note3\_2543():

...

1. # Opens the specified file
2. appuifw.app.content\_handler = appuifw.Content\_handler

(lock.signal)

1. appuifw.app.content\_handler.open") (u" http://www.dsmforum.org/events/DSM06")
2. lock.wait()

70

1. def Choice\_list():
2. # Popup menu Please choose
3. choices3\_2520 = [u"Registration", u"Program", u"Unregister"]
4. index = appuifw.popup\_menu(choices3\_2520, u"Please choose") 75 if index == None:
5. return ((call\_stack.pop()), False)
6. elif choices3\_2520[index] == u"Program": 78 return (SetForm, True)
7. elif choices3\_2520[index] == u"Registration":
8. return (Query3\_1481, True)
9. elif choices3\_2520[index] == u"Unregister":
10. return (Query3\_6300, True)

The generator first produces module import commands (lines 1–2) based on the services used, like the messaging module that provides SMS sending services. This is followed by documentation specified in the design. Next, each service and dialog is defined as a function. Lines 41–51 describe the code for the payment method selection that uses a List dialog. After defining the function name and comment, a variable named Payment is declared global to be available for the other functions in the application. Line 44 shows the list values as Unicode in a local variable, and line 45 calls the List dialog provided by the Python for S60 API.

Lines 53–55 create the function code for the welcome note. Lines 65–69 show the code for browsing the web. Lines 71–82 show the pop-up menu code produced by the generator defined in Listing 8.1.

Listing 8.5 shows a text message function, application exit, and common framework code for handling navigation stack and application start. Lines 111–120 are created based on the generator definition shown in Listing 8.3. Lines 113–115 define the global variables so that the text message function can access data entered in the design. Line 119 calls the imported SMS module and its sms\_send function. Parameters to the function, recipient number, message keyword, and content, are taken from the model and the generator takes care of forming the right message syntax. After all, it is always the same and is now defined just once in the generator.

The rest of the code is almost the same for all applications. Application stop (lines 122–126) is taken from the Stop state using the unique function name given by the generator as entering separate names in models for the Stop or Start states would just add extra modeling work.

Listing 8.5 Python code generated from Fig. 8.6.

1. def SendSMS3\_677():
2. # Sending SMS Registration
3. # Use of global variables
4. global PersonNamed
5. global Payment
6. string = u"Registration "\
7. +unicode(str(PersonNamed))+", "\
8. +unicode(str(Payment))
9. messaging.sms\_send("+358400648606", string)
10. return (Note3\_2227, False)

121

1. def Stop3\_983():
2. # Application stops here
3. global exit\_flag
4. exit\_flag = True 126 return (lambda: None, False)

127 128 def main():

1. old\_title = appuifw.app.title
2. appuifw.app.title = (u"Conference registration")
3. call\_stack.append(Stop3\_983) 132 state = Note3\_2543
4. try:
5. while not exit\_flag: 135 new\_state, add\_to\_stack=state() 136 if add\_to\_stack:

137 call\_stack.append(state) 138 state = new\_state 139 finally:

1. lock.signal()
2. appuifw.app.title=old\_title 142 appuifw.app.exit\_key\_handler = None

143 144 if \_\_name\_\_ == "\_\_main\_\_":

145 main()

Lines 129–130 save the old title and change the application name. In line 141, the old title is returned when exiting the application. The new application name is taken by the generator from the name of the design model. If alternative model naming should later become necessary, the metamodel could be changed to include a specific property for the application title.

The framework code takes care of the navigation stack. Line 11 in Listing 8.4 defines Call\_stack as a list. To ensure that there is at least one item in the list (if the user presses cancel immediately after starting the application) a Stop object is added to the stack in line 131. The first state (Note3\_2543 in the listing) is given and the application is started with the main dispatcher with the “try” line (133). The loop continues until exit\_flag is True; the Stop function sets the flag (line 125). In the loop, each function executed inside Python will always return a tuple with information about the next function and whether this current function needs to be stored on the stack. For example, Note (line 55) and SendSMS (line 120) have the parameter value False and they are not stored on the stack. In most of the other functions, add\_to\_stack is set to True, and a new state will be added to call\_stack.

When exit\_flag is True, lock.signal is included to ensure that the application is “awake” and closing goes smoothly. Finally, the old application title is returned and the application’s exit\_key handler is reset (there can be customized handlers during application execution). The last two lines show the standard way to start a Python application.

8.6 FRAMEWORK SUPPORT

During DSM creation, the S60 mobile phone platform and the Python for S60 framework were taken as given: The language creator simply made the generator call the services provided by the frameworks. An additional framework to support DSM was made to handle application start, exit, and navigation.

In addition to basic start and exit code, this domain framework included a dispatcher that started the application by calling the first function and looping until the application exit was reached. This code was generated as part of the main function of the application code, as already shown in detail with the application code in Listing 8.5. A bigger part of the framework code was made to handle the navigation stack, which enables getting back to the previously accessed function without starting a new one. Here, basic stack handling operations were designed so that application objects (functions in generated code) that could be targets for cancel navigation are pushed onto the stack after accessing the next element in the flow. Correspondingly, they are popped from the stack when they are the target of a cancel. Not all concepts in the navigation flow were put on the stack as they were not suitable for canceling. For example, a canceling operation that would trigger an already closed stand-alone application outside the Python application scope would not make sense.

The last piece of the framework code was made to take care of concurrency control in the UI application code. Python for S60 requires that execution paths be blocked when certain phone services are used: the concept of an active object in the underlying Symbian OS needs to be recognized. Otherwise, the execution control inside the Python forS60 application could notbe reached when control is given to the underlying phone service. The Python adaptation to Symbian automatically handled part of this thread management, but depending on the application logic, the Python programmer also needed to take care of concurrency by using the API commands available from the library. In the DSM solution, setting wait locks and releasing them in the correct manner was handled with the framework code. The code calling the concurrency library was put into the generator so the modeler did not need to think about it.

The connection between the framework code and the generated code could be established in many different ways. The framework code could be implemented in a

### MAIN RESULTS

library and called during generation. Further, the library could be part of the function library available in the modeling tool or saved to external files. To make generation safe and easy, the framework code was put into the generator. This way, the user would not be responsible for finding the correct function or need to take care that the external files containing the framework code were available for the generator.

8.7 MAIN RESULTS

The DSM language and generator were created to make development fundamentally easier and faster. Developers were not expected to master the details of the S60 architecture and coding: that unnecessary complexity was hidden with a language that fit better to the development tasks. We believe that the above few examples illustrate how the objective was achieved. Perhaps an even better indication of this is that, using the language, you too could now make mobile applications! See the book web site for instructions on using DSM solution.

Making application development easier was valued highly among mobile developers, especially those who had been using traditional manual coding approaches. The complexity of the underlying platform and mobile programming model served to further underline the benefits. A good indication of this is a statement made by a mobile application developer, Simo Salminen from Flander Ltd: “When you’re used to Symbian C++, it’s quite a shock to notice how easy UI application building can be.” This statement reveals the clear difference between programming on top of a platform and using DSM on top of a platform.

This increase in the level of abstraction away from programming constructs, and the use of code generators also lead to improved productivity. Although this improvement was never explicitly measured for large applications, it was evident. Markus Ha¨nninen from Enpocket saw that the DSM solution “greatly speeds up the development process and communication with end users.” This communication is usually found to even work better later when models expressed with higher level constructs become useful to people outside the development team too.

8.8 EXTENDING THE SOLUTION TO NATIVE S60 C++

As we mentioned earlier, S60 applications are normally made in C++, as are the native platform libraries that applications call. The C++ Software Development Kit (SDK) is widely regarded as being something of a nightmare to use, with baroque libraries and a number of nonstandard conventions. On the earlier Psion EPOC platform, these conventions were able to offer more efficient memory, power, and even CPU usage, but as the platform has grown, the resulting phones seem to have been unable to maintain these advantages. The result is a platform that demands a lot from the programmer: there are so many hoops to jump through and choices to make that writing concise, elegant code is all but impossible. In S60’s defense, most of today’s mobile phones do not allow third parties to write native applications at all. The growth of the platform can also be explained by the explosion in the kinds of functionality it must support. The original EPOC allowed simple personal productivity and network applications, but now there are also all the phone and camera functions with their underlying protocol stacks and drivers.

8.8.1 Extending the Domain

The Python support only exposes a small fraction of the native libraries, focused on the dialog-based UI. While this is sufficient for an interesting subset of applications, it leaves out the richer tabbed views and menus that are used in many even slightly larger applications. The Python DSM solution also lacks the ability to support localization and build entirely new first-class applications that are run independently of other applications (like the Python interpreter). When extending the DSM solution to generate C++, we decided to extend the domain it covered to include these concepts of view, menu, and application. The original diagrams would describe what happened in a single view, and a new top-level diagram would specify the application and its views and menus.

Figure 8.9 shows an example of a top-level diagram based on an S60 SDK sample application for demonstrating menus. The top row contains a simple Application object, ConfApp, that specifies the name and unique ID of the application. The second row specifies the three tabbed views of the application, with the initial default view starred. Each view can explode to a diagram specifying the dialogs used in that view, e.

g., here, the first view from the sample has been changed to point to the conference registration diagram defined earlier. The third row contains the menus used in the views. In S60, the menus are specified as if they were in a menu bar containing toplevel items like System, App, View, and Context. Current S60 phones, however, only display one simple menu, formed by concatenating the menus for these top-level items. A menu item may point to a custom command (not yet implemented in this

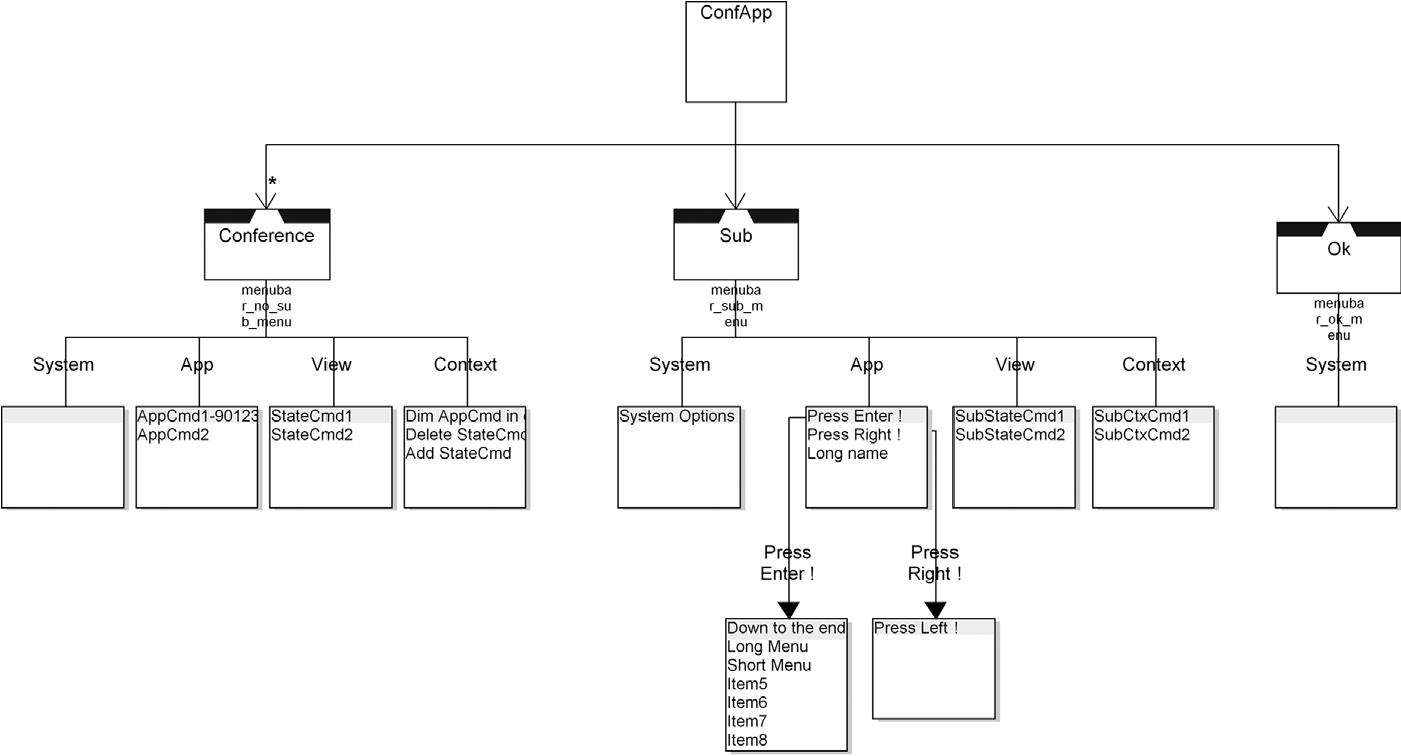


FIGURE 8.9 S60 application, views, and menus

EXTENDING THE SOLUTION TO NATIVE S60 C++

DSM solution), a system command like Copy or Quit, or a submenu. Two example submenus are shown in the fourth row.

8.8.2 C++ Generators

Even a simple S60 application like Fig. 8.9 requires the developer to create a large number of files—27 in this case. The architecture of these files seems to tend toward high coupling and low cohesion, effectively preventing reuse of any of the files in a subsequent application. The sheer number of files makes building a generator somewhat laborious, but only in the same way as building a single application by hand would be. The verbosity and repetition between the files do, however, mean that a small amount of modeling effort will result in a prodigious amount of output in terms of code.

The structure of the C++ generators is shown in Fig. 8.10. The generator begins on the left by creating the five directories into which the files must be created, and an Autobuild batch file that will kick-start the S60 build process and emulator. We will look at the main body of the generation from the top of the figure down, following the grouping into numbered gray areas. Group 1 handles the generation of the static user interface description from the top-level diagram: the views, menus, and reusable strings. Its \_RSS generator produces the single largest file, AknConfApp.rss, weighing in at an impressive 16 KB. Separate \_RSS\_AIF and \_RSS\_caption generators add some smaller RSS files for the application as a whole. Group 2 contains some small generators that descend to the subdiagrams of the views, adding to the main RSS file the strings used for choices in Pop-up Menus and Queries.

Group 3 creates the HRH file, which contains an enum of all the localizable strings used in the application, and the LOC file, which defines the English language version of each string. In the models, each string used is represented as both a unique name and the English language string, the latter being displayed in the diagrams for ease of reading. The MMP, PKG, and INF files and corresponding generators produce the application descriptor and make files.

Bizarrely, the same variable or function name must be formatted in different ways in different places: lowercase in RSS files and uppercase in some places in C++ files. There are also conventions for prefixes for names in various places. These requirements are handled in the short utility generators in group 4, which are called by many of the other generators.

So far, none of these files has actually been real C++ code: some had their own S60-specific syntaxes, some were simple C++ definitions. The C++ code makes up two thirds of the total size and number of files and is created by groups 5, 6, and 7. In S60, an application is represented by three C++ files: an application, UI, and document file, and these are created by group 5. Each view has its own file and an ancillary container file, and these are created by group 6. The dialog contents of each view are created as functions in its file by group 7, which has a subgenerator for each dialog type.

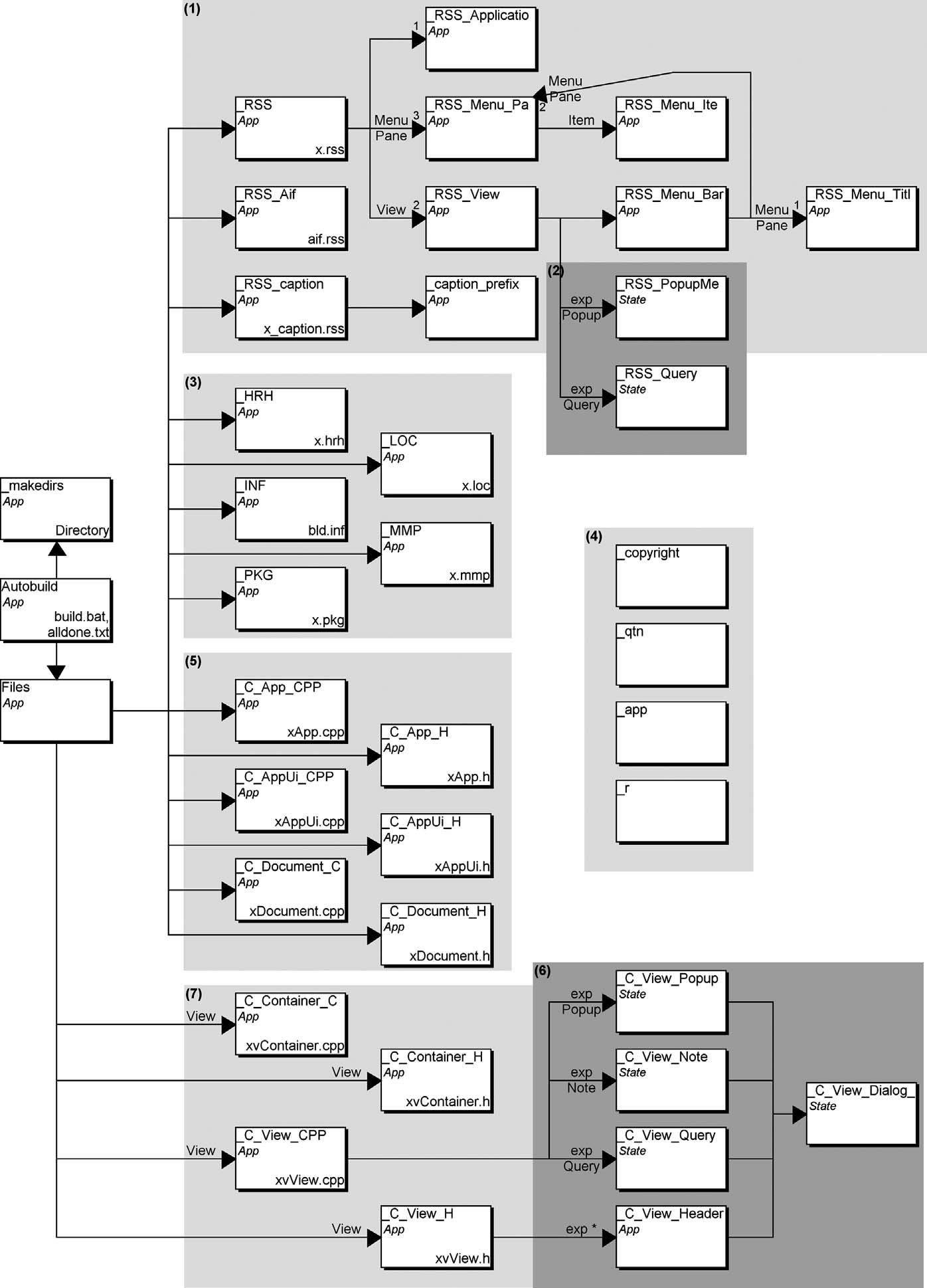


FIGURE 8.10 Structure of S60 C++ generators

### SUMMARY

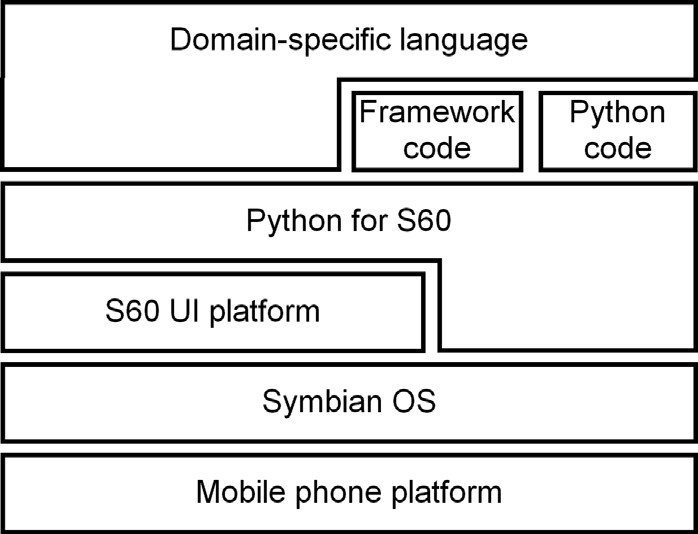


FIGURE 8.11 Domain-specific language on top of platforms

8.9 SUMMARY

The DSM case described here shows how the level of abstraction can be raised on top of an existing platform—or rather, on top of several layers, as illustrated in Fig. 8.11. First, there is a mobile platform providing some hardware-related software. On top of this, an operating systems provides a set of libraries and basic mobile services and applications. Next, to make application development faster and easier, the level of abstraction is raised by providing a suite of libraries and standard applications with a UI application platform. In our case, the UI platform has been S60 (Babin, 2005). On top of this UI platform, the most typical application domain, enterprise, and administrative applications are selected, and a framework is provided based on Python. The DSM solution is made to further hide application development complexity to make it easier and faster.

We have focused here mostly on creating a DSM solution for Python for S60. To make applications, the solution provides a DSM language backed by framework code to support the generated code. The modeling language is based on UI concepts and follows the mobile phone programming model with the use of dialogs, UI controls, and phone services. Each of these concepts is used during code generation to apply the services from the Python for S60 framework. A few language concepts were also used to add manual code that does not relate directly to the Python framework but allows Python programming. The need for manual programming was reduced to a few places where higher level constructs than Python were not found.

The case also shows how DSM can evolve when the underlying platform evolves. During the case, Python for S60 had three releases, yet no changes were needed to the models describing the applications. The changes made by Nokia to the Python for S60 framework—mostly new services—led to changes in only the language and generator. For example, when the text message sending module was restructured and its API changed, only the code generator needed to be changed. Old applications using the text messaging could be updated to the new version simply by generating the code again. This is in contrast to manually changing all hand-coded applications that used text messaging.

The DSM solution can also evolve in the future. After having made enough similar kinds of applications, perhaps a review of manually written Python code included in models will detect similarities that can be abstracted to the modeling language. Also, if Python applications are built in other areas where S60 UI styles can’t be used, like games, then the possibility of adding more Python code can be provided within the DSM solution. Most likely it would be better to provide libraries for graphics and extend the DSM solution to cover them too. Then those newer application areas could have the same benefits that the current DSM solution provides.