Defining Generators with MetaEdit+

MetaCase Document No. DG-5.5

3rd Edition, Jan 2020

MetaCase
Ylistönmäentie 31
FI–40500 Jyväskylä
Finland

Tel: +358 400 648 606
E-mail: info@metacase.com
WWW: https://www.metacase.com

Contents

Preface

Developers usually agree that it does not make sense to write all code character-by-character. After coding some features, we usually start to find similarities in the code, patterns that seem to be repeatable. Because nobody wants to write routine code, the idea of code automation — generating the repeatable portions of code automatically — arises both naturally and repeatedly.

We introduce in this paper how code generators work and how they can be defined in MetaEdit+. The appendix describes some typical cases of code generators in more detail, covering various generator approaches and various target languages (XML, Python, C and Assembler). Each example produces the application logic and behavior, not just the easier-to-create static structures or skeletons.

For executing and exploring these generators you should use MetaEdit+ Workbench or the evaluation version, which is available for download from https://www.metacase.com. For further information about MetaEdit+, please refer to the 'MetaEdit+ Users Guide', 'MetaEdit+ Workbench Users Guide' or our web pages at https://www.metacase.com.

1 How Generators Work

A code generator is an automaton that performs three things:

1) Access and navigates in the models

2) Extracts information from them

3) Transforms it to an output in a specific syntax

Let's next inspect these steps in more detail using a state transition diagram as an example. Figure 1 shows a partial state machine specifying reading of a payment card. The model has a start element, two states and two transitions.

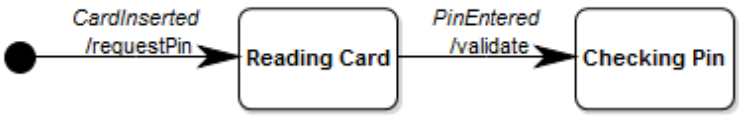

Figure 1. A partial state transition diagram

1.1 ACCESSING MODELS

Access of models is based on the metamodel of the language: A generator can access and navigate models based on the concepts used in the modeling language. Figure 2 shows a metamodel describing the concepts of state machines – all used in model shown in Figure 1.

Now the generator can start navigation based on a certain root element, like 'Start' object, or seek for certain object types (e.g. 'State' object) or be dependent on the various relationship types (e.g. 'Transition' relationship).

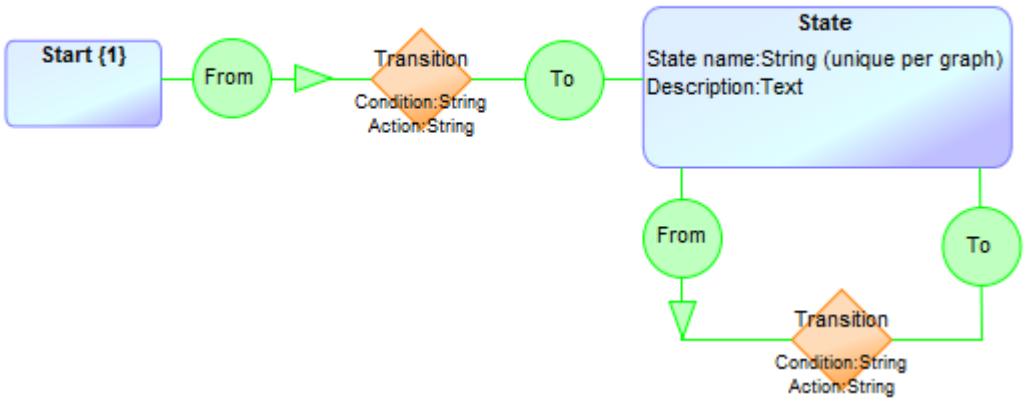

Figure 2. A metamodel of the state transition diagram used in Figure 1

A generator accessing and navigating models based on a start state can be defined in MetaEdit+ as follows:

```
1 foreach .Start {
2   do >Transition.State {
3        /* generator handling the accessed state here */
4   }
5 }
```

Note that the generator script uses the terms defined in the metamodel, such as 'Start', 'Transition' and 'State'. Since the metamodel states that there can be only one start state (see 'Start {1}' in the metamodel) the generator always finds the one and only possible start state. If such start object is not found from the model the above defined generator does not do anything. Next in line 2 is the navigation following the transition: the generator refers to the 'Transition' relationship defined in the metamodel to access the first state.

MetaEdit+ Generator Editor allows you to write and debug the generators. While writing the generator you can directly use the concepts of the modeling language (metamodel) as well as trace from the generated code back to the models. This makes generator development fast and easier. Figure 3 shows the Generator Editor for the above shown generator script. For details of Generator Editor and Debugger see Workbench User's Guide.

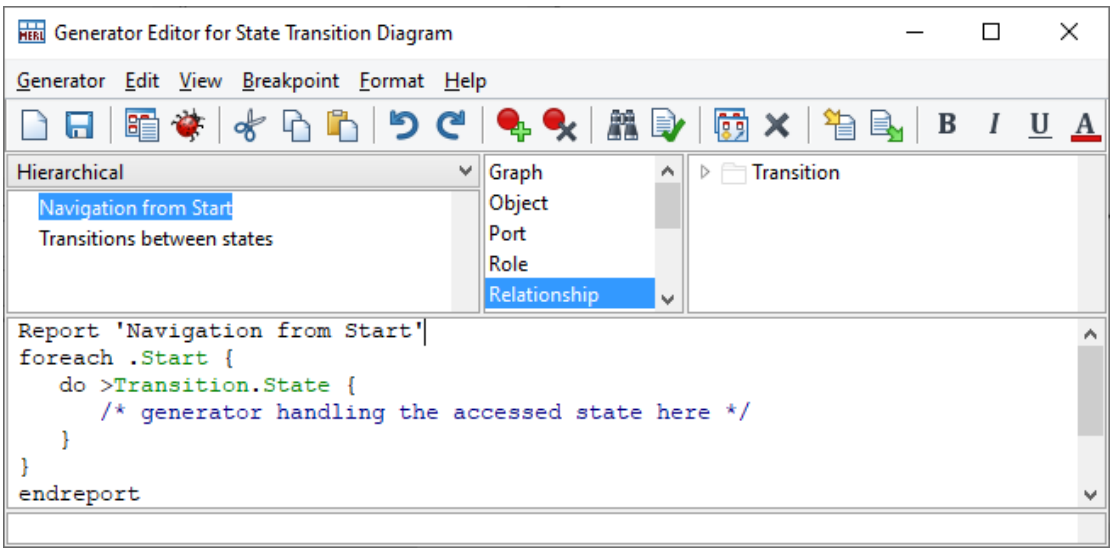

Figure 3. Generator script in MetaEdit+ Generator Editor

Rather than accessing models from 'Start' element we could instead write a generator that navigates models by following all the states like:

```
foreach .State {}
```

If we would like to access models based on transitions the generator would be respectively:

```
foreach >Transition {}
```

Even more navigation choices are available if the generator definition expects certain value in the model for navigating and accessing model elements, such as access only transitions which have a condition defined:

```
foreach >Transition; where :Condition; {}
```

Or access states which have substates defined in another (sub)diagram:

```
foreach .State {
    if decompositions; then
         /* generator for the state with a substate here */
    endif
}
```

These options would allow generator to navigate connections or submodels based on the depth or the breadth first or apply some order in navigation and access.

While usually the data for accessing and navigating in models is already stored into the model elements also additional information can be used. These can include:

- Sorting order (e.g. states in alphabetical order), uniqueness (e.g. exclude duplicates), etc.
- Spatial location of model elements, e.g. size, location in relation to other elements, if an element is inside another element, etc.
- Various model administration data, such as creation time, version, author information etc.

If there are multiple models based on different languages, the navigation can be based on the same principles than accessing just one model. Links between the models can be identified by the code generators, like using the subdiagram access described above, find elements that are shared among different models, apply string matching, or use reference elements to link the models.

1.2 EXTRACTING MODEL DATA

Extracting model data is again based on the metamodel. The code generator retrieves information from models as metamodel allows storing them. In the simplest case, each modeling element produces certain fixed code that includes the values given in the model. A typical example of this is a template-based code generation, like:

```
1 foreach .State {
2   'case '
3   :State name
4   ': ' newline
5 }
```

Line 2 in the generator produces fixed text (‘case’) and line 3 extracts the name of the state from the currently accessed state object. Line 4 adds ‘:’ and a newline character. When this generator is run for the model described in Figure 1 it produces a line for each state based on the template given in the generator as follows:

```
case Reading card:
case Checking pin:
```

The generator can combine navigation and model access like in:

```
1 foreach .State {
2   :State name;
3   :Documentation;
4   do ~From>Transition {
5        :Trigger
6   }
7 }
```

Here lines 1 and 4 specify how models are navigated and lines 2, 3 and 5 access information from the model. By navigating and accessing the models different kind of generation results can be obtained, like producing the code following sequential information, function calls, switch-case structures, transition tables etc. For details of the Generator Language see MetaEdit+ Workbench User’s Guide.

Rather than having one single generator script, generators can be modularized. For example, a generator called ‘_State’ can produce the state specific code, whereas ‘_Transition’ the code for triggers and actions, a third generator can do the navigation etc.

1.3 TRANSFORMING MODELS TO OUTPUT CODE

While navigating in models the data accessed is combined for the purpose of code generation (or for configuration script, document, metric etc). Generator can add additional information for the output as well integrate with the framework code or make calls to the underlying target environment and its libraries.

The generator may also transform the data retrieved from the model in the format required by the generated target language. For example, line 3 in the generator shown already earlier has now a translator '%var' that translates the name given in the model for a legal variable name.

```
1 foreach .State {
2   'case '
3   :State name%var
4   ':' newline
5 }
```

The translation '%var' removes the spaces from the state name with underscore, and the generated result is now:

```
case Reading_card:
case Checking_pin:
```

To summarize, the generator depends on and is guided by the metamodel, and by the input syntax required by the target environment. Appendix shows four different kinds of generator cases – producing different kind of code in different languages (Assembler, C, Python and XML).

2 How to Implement Generators

Before we can build a generator, we need to know what we want to generate. For this purpose we need a reference implementation; a working example including both a model and the output to be generated.

2.1 REFERENCE IMPLEMENTATION: MODEL AND ITS CODE

To get the code, a good practice is to ask the most experienced programmers to come up with the reference code. It's important to ask them to write in the same style as they would like to teach other developers; otherwise the code may include too many special tricks for particular cases, rather than good standard code that can be generalized to all applications in that domain. Generating that code makes a good impression. Even if you later abandon the generator for some reason, you still end up with standardizable and generalizable expert code. Using code from experienced developers also speeds up the generator creation: it simply requires less discussion in the whole team about different coding practices and standards.

Once you have the reference code, then model the equivalent application functionality with your domain-specific modeling language. Such a model might have one element of each of the three or four main modeling concepts. It could be in the beginning as small as in Figure 1, and later you can extend it by adding more elements. While creating the model check that it can give you the information you need.

2.2 A SIMPLIFIED PROCESS FOR BUILDING GENERATORS

The information in the model is distributed over objects, relationships and properties. The output has the same semantic content, plus some content that is related to the output language syntax, and some content that the generator will add. Clearly, the lion’s share of the variability in the output will come from the model: the generator will be the same for all models, and its output for a given model will always be the same, so it cannot add any variability to the output that is not found in the model.

You can thus look at the output and identify which parts of it are fixed text that will always be present, and which parts are simply values from the model. Between these two extremes lies the work for the meat of the generator: parts that are included or left out depending on some information from the model, and parts that are repeated for each of a given structure in the model.

These four kinds of parts cover the entirety of most generators, even for the most complex systems.

To cope with all possible generators, we need to add the possibility of massaging the values from the models - normally as a concession to the syntax of the output language. For instance, most languages require that names be composed only of alphanumeric and underscore characters, yet we may want to allow the names in the model to contain spaces – as done with %var translator earlier with state name.

A simplified process for building a generator is thus:

1) Paste the desired output code as the entire content of the generator.
2) Reduce each repeated section in the output code into one occurrence, with a generator loop that visits each model structure for which the section should occur.
3) For sections that have one or more alternative forms, surround them with generator code that chooses the correct form based on a condition in the model.
4) Replace those parts of the output that match property values in the model with generator code that outputs those property values, filtered as necessary.

In practice, steps 2 - 4 are often best performed in parallel on each chunk of output in turn.

2.3 STRUCTURING GENERATORS

In many ways, writing a generator is just a special case of writing a program: there are as many different ways to structure the same behavior as there are programmers. Some ways have, however, been found to be better than others: easier to create, debug, maintain and understand. Many of these ways of structuring code are also appropriate when writing generators and the good developer will naturally use them. Generators also have some special requirements of their own, mainly because of the strong existing structure provided by the modeling language.

Usually the generators are structured into smaller ones. Having a hierarchy of generators, each calling other sub-generators, work particularly well. Here a top-level “autobuild” generator is thus visible to the user, and is responsible for building the resulting application. This generator calls other generators and these split their task up according to the various files to be produced from each model. Where necessary and possible, the file generators again split the task up according to the object types in the model. If more than one programming language is needed for the same set of models, this structure can be built in parallel for each.

One particularly good way to structure generators is to define them for each individual modeling language, and each main language concept could have own subgenerator. For example, there could be one generator that takes care of ‘States’ another producing the code for ‘Transitions’ etc. The obvious benefit is that if the modeling language changes you may easily find the part of the generator that needs to be changed too. You may also structure the generators based on the files to be generated, or based on sections in a file to be generated.

2.4 GENERATING OTHER THAN CODE

Code generation is not the only place for automation. You can make generators which produce configuration information, test cases, documentation and data dictionaries, check the consistency of the models, create metrics, analyze model linkages, and export models to other programs, such as simulators, compilers, emulators etc. The advanced scripting commands allow you to print or save designs in various formats, output to multiple files, and call external programs. Having a single source and multiple targets can be very beneficial, because when making modifications, developers need to make a change in only one place, and the generators produce the needed outputs.

3 Summary

Building a generator is about saying how to map model concepts to code or other output. In the simplest case, each model element produces certain fixed code that includes the values entered into the symbol as arguments by the modeler. To go a bit further a generator can also take into account relationships with other model elements or other model information, such as submodels, models made with other languages, or pre-existing library code. The way the generator takes information from models and translates that into code depends on what the generated output should look like.

MetaEdit+ provides tools for defining and debugging generators. With Generator Editor you can choose which parts of your design data in the models you want to have written out and in which format. The generator definition can then be used multiple times by multiple developers to produce code from various design models.

The appendix shows examples of various generators in more detail. You may also investigate the tutorials in the evaluation version of MetaEdit+ available to download at http://www.metacase.com/download. In particular the family tree tutorial shows a case of HTML generation, and Digital watch tutorial shows Java and C# code generators.

APPENDIX: Examples of code generation

It's useful to look at some real-life cases where to generate code. While there's insufficient room in this paper to cover everything, the example cases should suffice to show that code generation isn't just for one particular niche. The cases cover various generator approaches and various target languages, including XML, Python, C and Assembler. Each example produces working code and contains both the application logic and behavior, not just the easier-to-create static structures.

EXAMPLE 1: GENERATING XML

Defining generators to produce XML is often quite a straightforward process, especially if the modeling language maps well onto the XML schema. This is in fact normally the case, since both are designed to be a good way of describing the same domain. Where XML schemas have had to sacrifice understandability to cope with the limitations of XML, the modeling language can do things in a more natural way. In this case, the generator will do a little extra legwork to produce the verbosity and duplication required in XML.

This concrete example uses the Call Processing Language (CPL). A guide on this example is available in manuals and you may find this example from 'CPL' project of MetaEdit+ demo repository. CPL describes and controls Internet telephony services, in particular, specifying how a system routs calls. The structure of the CPL language maps closely to its behavior, so any CPL service developer can easily understand and validate the service from these graphical models (see sample in Figure 4). Figure 4 also shows the benefit of models; even a non-CPL developer can largely understand the model, whereas deciphering the same model in XML would be considerably harder.

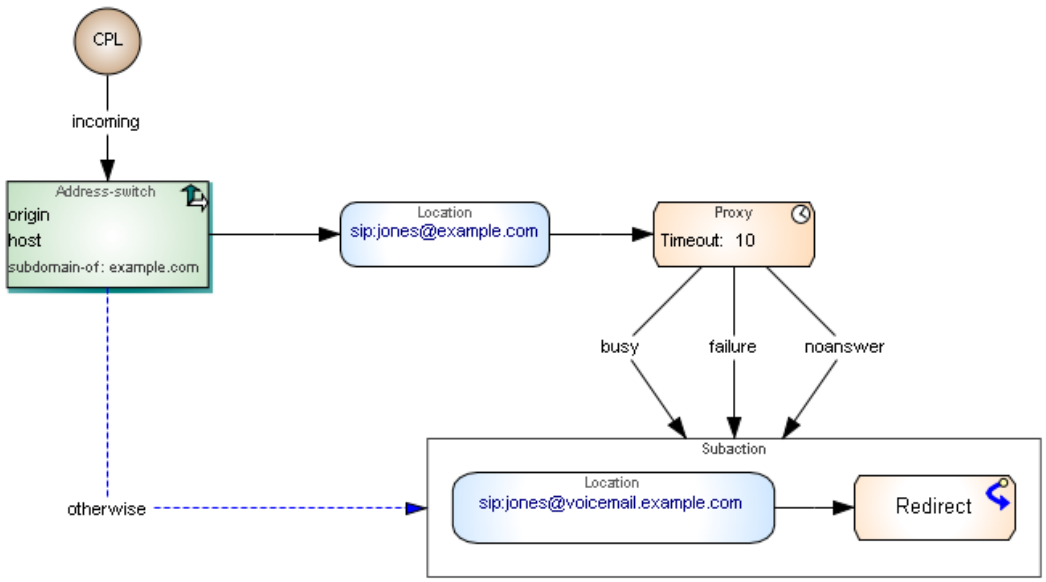

Figure 4. Redirecting Calls: A sample call redirecting service expressed in CPL

The modeling language includes concepts such as proxy, location, and signaling actions, which are essential for specifying IP telephony servers. The generation process is very clear-cut, because the XML already defines the language concepts, and the property values of the modeling constructs are attributes of the XML elements. The task of the generator is to follow the nodes via their connections and create the corresponding CPL document structure in XML. The following code shows the generator output when producing XML from the specification illustrated in Figure 4.

```
01  <?xml version="1.0" encoding="UTF-8"?>
02  <!-- DOCTYPE call SYSTEM "cpl.dtd" -->
02  <!-- DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx CPL
    1.0//EN" "cpl.dtd" -->
03  <cpl>
04  <subaction id="voicemail">
05    <location url="sip:jones@voicemail.example.com">
06      <redirect />
07    </location>
08  </subaction>
09  <incoming>
10    <address-switch field="origin" subfield="host">
11      <address subdomain-of="example.com">
12        <location url="sip:jones@example.com">
13          <proxy timeout="10">
14            <busy><sub ref="voicemail" /></busy>
15            <noanswer><sub ref="voicemail" /></noanswer>
16            <failure><sub ref="voicemail" /></failure>
17          </proxy>
18        </location>
19      </address>
20      <otherwise>
21        <sub ref="voicemail" />
22      </otherwise>
23    </address-switch>
24  </incoming>
25  </cpl>
```

The generator starts by going through all the subactions of the service specification. This example contains only one subaction, the voicemail box at the right bottom corner of the model, for which the generator produces lines 4—8.

```
04  <subaction id="voicemail">
05    <location url="sip:jones@voicemail.example.com">
06      <redirect />
07    </location>
08  </subaction>
```

This "voicemail" subaction defines a location element (line 5) as well as a redirect element (line 6), which activates the redirection automatically.

After producing subactions the generator starts to specify the main call processing actions. It goes through the service specification from a service start (the brown "CPL" circle in Figure 4). The generator crawls the connections from the CPL circle through the "Incoming" relationship to the Address-switch object. It produces the properties of the Address-switch node as element attributes in the generated output (lines 10—11).

```
10    <address-switch field="origin" subfield="host">
11      <address subdomain-of="example.com">
```

The generator continues to follow the main flow path arrow to the next object and produces the location definition (line 12).

```
12        <location url="sip:jones@example.com">
```

The path continues and the proxy handling is generated on lines 13—17, first the timeout attribute (line 13) followed by three alternate connections from the proxy element.

```
13          <proxy timeout="10">
14            <busy><sub ref="voicemail" /></busy>
15            <noanswer><sub ref="voicemail" /></noanswer>
16            <failure><sub ref="voicemail" /></failure>
17          </proxy>
```

Finally, the generator produces lines 20—22 for the cases where the call origin has an address other than example.com.

```
20      <otherwise>
21        <sub ref="voicemail" />
22      </otherwise>
```

The generated code forms a complete service whose validity has already been checked at the design stage. Because the modeling language contains the rules of the domain, service creators can create only valid and well-formed design models. The modeling language can also help service creators with consistency checks and guidelines, for example by informing them about missing information (such as the lack of a call redirect specification). These rules are highly domain-specific and thus can be handled only with a domain-specific language.

EXAMPLE 2: GENERATING ASSEMBLER FOR 8-BIT MICROCONTROLLERS

You can apply the same model-navigation approach in other cases, too. Here's a slightly more demanding case—generating 8-bit code for an embedded device. This particular device has a voice-menu system that enables remote control of many home-automation features, such as turning a light on or off, setting the room temperature, air conditioning control, etc. The system is programmable via an 8-bit microcontroller using an assembler-like programming language. In this case, code generation must take into account issues such as code size and memory usage that were not relevant when producing the XML in the previous example. You may find this example from 'Home automation' project of MetaEdit+ demo repository.

Besides the basic operations such as accessing memory addresses, calculation, comparison and jump, the developer must also handle a few operations specific to voice menus, such as reading menu items aloud. The domain-specific language includes these concepts directly in the modeling language. The model in Figure 5 illustrates an example of such a language.

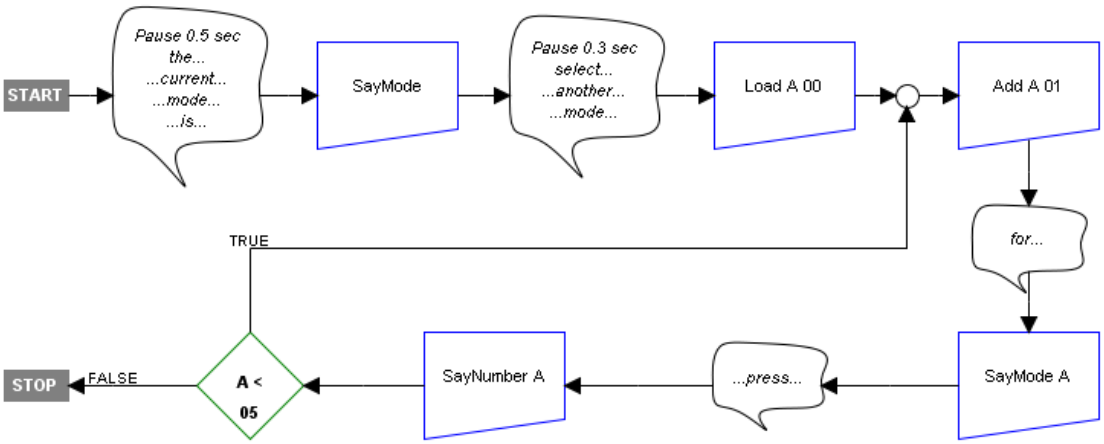

Figure 5. Setting Parameters: A design for setting one parameter in a voice menu system

This model and example code uses names to denote the variables, but in production, an organization would substitute real memory addresses. Here's a sample of the code generated from the model in Figure 5.

```
01    Speak 0x01 (Pause for 0.5 sec)
02    Speak 0x02 (the...)
03    Speak 0x03 (...current...)
04    Speak 0x04 (...lifestyle...)
05    Speak 0x05 (...is...)
06    GetLifeStyle
07    Speaks Lifestyle
08    Speak 0x06 (Pause 0.3 sec)
09    Speak 0x07 (select...)
10    Speak 0x08 (...another...)
11    Speak 0x04 (...lifestyle...)
12    FillMemB 00
13  :3_844
14    Add to MemB 01
15    Speak 0x09 (for...)
16    Speaks Lifestyle
17    Speak 0x10 (...press...)
18    Speak number MemB
19    Is MemB >= 0F
20    IFNot
21    Goto 3_844
```

The fundamental code-generating solution relies on a flow-like execution of actions specified in the model. The metamodel for the language is relatively simple. Each elementary operation type (speaking, memory access, and comparison) has been implemented as an individual modeling concept, while the flow of execution and conditional jumps appear as relationships between these concepts. Each modeling concept also carries information about related design attributes such as the command tag, possible parameters, or conditions. As the assembly language here is domain-specific, with one assembly mnemonic reflecting one basic voice menu operation, the mapping from modeling concepts to mnemonics is solid and simple.

The generator follows the flow of operation via relationships, transforming the information contained in each design element into output code. For example, the generator uses the first bubble on the top left corner of diagram to generate the set of speech commands in lines 1—5.

Because the spoken audio samples consume a lot of memory, each word is stored only once and reused when constructing complete sentences. Thus, the spoken messages in menu elements consist of a sequence of individual words or short phrases, along with the memory addresses for the corresponding audio samples. When the code generator meets such a construct, it simply iterates over the collection and outputs the speak command for each word followed by the address of the sample. When the same audio appears multiple times the generator outputs the same memory address—for example, lines 4 and 11 use the same memory address.

Variations of the speak command can read out variable values (see lines 7 and 16) or numeric arguments (see line 18). In addition to speech commands, there are also operations for getting (line 6) and setting (line 12) values of variables.

You can see a slightly more complex part of the generation process in lines 12—21, which reads out the selection numbers for all predefined lifestyle settings.

```
12    FillMemB 00
13  :3_844
14    Add to MemB 01
15    Speak 0x09 (for...)
16    Speaks Lifestyle
```

```
17   Speak 0x10 (...press...)
18   Speak number MemB
19   Is MemB >= 0F
20   IFNot
21   Goto 3_844
```

First, it initializes the selection variable memB (lines 12 and 14). Then it retrieves and speaks the information about a predefined lifestyle (lines 15—17, simplified here to save space) and the selection option (line 18). Lines 19—21 represent the conditional jump. If the selection variable memB is less than 15 (0F hex), the code jumps back to line 13 and continues with the next predefined lifestyle, repeating the loop until memB equals 15.

In this case the key to the successful code generation is the modeling language. While there are a few domain-specific commands included in the target language itself, there is no real framework on the platform side to make the generation easier—and as you can see, the generation process itself does not include any "magic." The model level covers the structural and behavioral essence of the system, leaving no need for the generator to tackle complex variation or implementation issues related to the low-level language used.

EXAMPLE 3: GENERATING PYTHON FOR MOBILE PHONES

Navigating models based on connections is just one possible way of translating models to code. To illustrate other ways generators can work here's a case that generates smart phone applications by producing function definitions for individual model elements. A guide on this example is available in manuals and you may find this example from ‘S60 Phone’ project of the MetaEdit+ demo repository.

In this example, the underlying phone platform provides a set of APIs and expects a specific programming model for the user interface. To enable model-based generation, a specification language and generator follows the programming model and APIs. Figure 6 shows a sample design.

The modeling concepts in Figure 6 are based directly on the services and widgets that smart phones offer for application development. Modelers describe the behavioral logic of the application mostly based on the widgets' behavior and on the actions provided by the actual product. If you are familiar with some phone applications, like phone book or calendar, you can probably understand what the application does by studying the model.

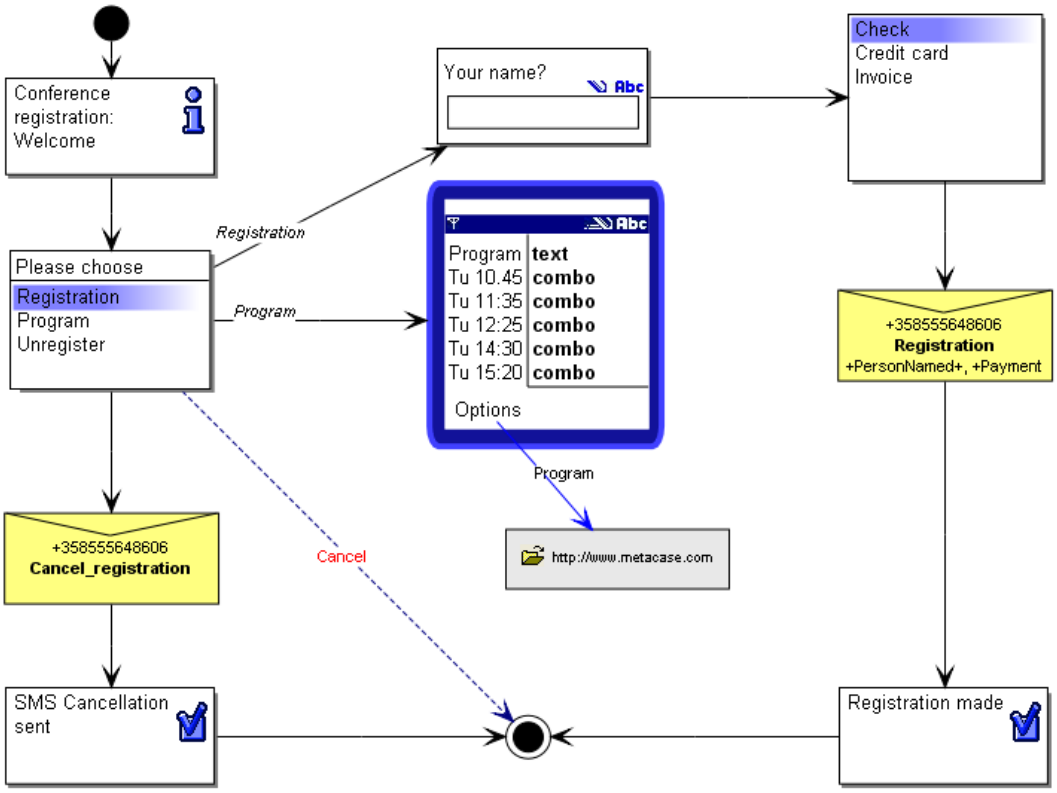

Figure 6. Conference Application: The figure shows the design for a conference application intended to run on a smart phone

From the design in Figure 6, the generator produces function-based code that the target device or an emulator can compile and execute. The generator itself is structured into modules, one generator module for each modeling concept. For example, one generator module takes care of lists, another handles confirmation dialogs etc. Because several concepts require generation of similar code, such as flow of control to the next concept, parts of the generator definitions are made into subroutines used in several places. The generator includes some framework code for dispatching and for multi-view management (different tabs in the pane). Here's some sample Python code generated from the design in Figure 6.

```
01  import appuifw
02  import messaging
03
04  # This app provides conference registration by SMS.
...
33  def List3_5396():
34  # List Check Credit card Invoice
35      global Payment
36      choices3_5396 = [u"Check", u"Credit card", u"Invoice"]
37      Payment = appuifw.selection_list(choices3_5396)
38      if Payment == None:
39          return Query3_1481
40      else:
41          return SendSMS3_677
...
85  def SendSMS3_677():
86  # Sending SMS Conference_registration
87  # Use of global variables
88      global PersonName
```

```
89      global Payment
90      string = u"Conference_registration "\
91            +unicode(str(PersonName))+",  "\
92            +unicode(str(Payment))
93      messaging.sms_send("4912345678", string)
94      return Note3_2227
...
101   def Stop3_983():
102   # This applications stops here
103      return appuifw.app.set_exit
...
107   f = Note3_2543
108   while True:
109      f = f()
```

The generator starts by outputting module import statements (lines 1—2) based on the services used. First, the overall application UI framework, and then—because the model contains SMS actions (yellow envelope symbols)—the messaging module. The comment following the import statements is simply taken from the documentation entry specified in the design. Next, the code defines each service and widget as its own function. Rather than producing functions in an arbitrary order, the generator produces functions arranged by type, for example, all list functions followed by SMS/text message functions etc.

Lines 33—41 describe the code for the payment method selection that uses a list widget. After defining the function name and comment, the code declares a global Payment variable. Line 36 shows the list values as Unicode in a local variable, and line 37 calls the List widget provided by the framework. The code handles sending SMS messages (lines 85—94) in a similar way to the List widget. Line 93 calls the imported SMS module's sms_send function. The generator takes the parameters to the function (recipient number, message keyword and content) from the model, and handles forming the right message syntax. These messages are clearly defined, always using the same pattern.

The end of each function includes code to call the next function based on user input. For sending an SMS, the generator simply follows the application flow (line 94), but for list selections, the situation is little more complex. Depending on the user's selection from the list, different alternatives can exist. While in this case the code always takes a single path forward regardless of the value chosen from the list, there is the implicit possibility of a cancel operation (pressing the Cancel or Back button). The generator automatically creates operation-cancelling code to return execution to the previous widget (lines 38—39), or if a choice was made, moves on to send the SMS (lines 40—41). Having the DSM implicitly handle Cancel or exception actions is just one extra way to make describing applications simpler. In other words, the modeler doesn't need to do anything in most cases, only specifying behavior that differs from the default (such as the diagonal Cancel relationship to the end state from the first menu).

In the final function, the generator creates application exit code based on the application's end state (see lines 101—103). Finally, a dispatcher starts the application by calling the first function (line 107). This function, like all the others, returns the next function to call, and lines 108—109 handle calling the next function after each function has finished. Handling the calls this way, rather than having each function itself call the next function, provides a kind of tail recursion to reduce stack depth when moving from one function to the next.

EXAMPLE 4: GENERATING C FOR DIGITAL WRISTWATCH APPLICATIONS

The last example illustrates C code generation from state machines. There are several ways to implement a state machine code and the best solution for a given situation depends of course on such things as the target language and platform, requirements for the size and efficiency of the code, etc. You may find this example from 'Digital Watch' project of MetaEdit+ demo repository and a tutorial in manuals describing also Java and C# code generation.

The digital wristwatch example's architecture consists of a set of applications for displaying the current time, stopwatch, alarm etc. The modeling language focuses on capturing the static elements of the watch application, such as buttons, time units, icons, etc., and on capturing the functionality using a kind of state machine. It extends the traditional state transition diagram semantics with domain-specific concepts and constraints. For example, each state has a link to a display function that presents the time, state transitions are triggered only by pressing one button once, arithmetic operations are limited to those relevant for operating on time units, etc. Figure 7 is an example application model for showing and editing the current time, and related operations. Listing contains the corresponding generated C code, slightly abridged to save space.

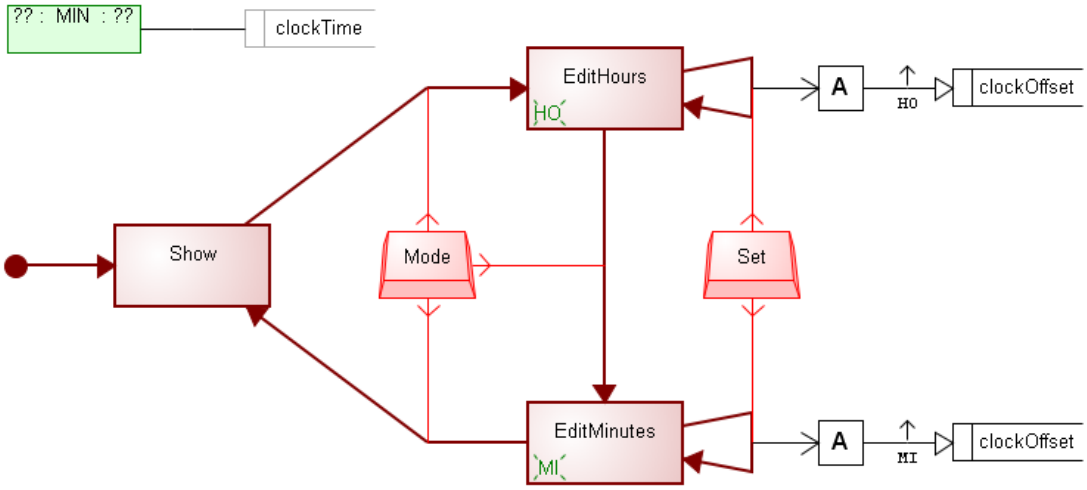

Figure 7. Watch Application Design: The figure shows the design for a watch application that can display and edit the current time

```
01 typedef enum { Start, EditHours, EditMinutes, Show, Stop }
   States;
02 typedef enum { None, Mode, Set } Buttons;
03
04 int state = Start;
05 int button = None; /* pseudo-button for following buttonless
                        transitions */
06
07 void runWatch()
80 {
09   while (state != Stop)
10   {
11     handleEvent();
12     button = getButton(); /* waits for and returns next button
                                press */
13   }
14 }
```

```
15
16 void handleEvent()
17 {
18   int oldState = state;
19   switch (state)
20   {
21     case Start:
22       switch (button)
23       {
24         case None:
25           state = Show;
26           break;
27         default:
28           break;
29       }
30     case EditHours:
32       switch (button)
33       {
34         case Set:
35           roll(clockOffset, HOUR_OF_DAY, 1, displayTime());
36           break;
37         case Mode:
38           state = EditMinutes;
39           break;
40         default:
41           break;
42       }
43     case EditMinutes:
...
55     case Show:
...
64     default:
65       break;
66   }
67   button = None; /* follow transitions that don't
                       require buttons */
68   if (oldState != state) handleEvent();
69 }
```

Defining the mapping process from model to code is reasonably painless. The generator goes through the design and creates enumerations (enums) that will act as unique labels for states and buttons (lines 1—2). These are then initialized in lines 4—5. Next the generator outputs a boilerplate runWatch() function, common to all applications. For each input event, runWatch() invokes the main behavioral part of the application, handleEvent() (lines 16—66). In handleEvent(), what to do and where to go to next depends on the preceding state, and what the input event was. The generator thus reads the state transition diagram and implements it using a simple nested switch statement. The generator produces this by iterating over the states in the model; for each state it iterates over the transitions leaving that state.

A transition can also trigger actions. The example model needs to support only basic time units for arithmetic operations. Setting time variables is a simple assignment; but rolling a digit pair up or down is actually somewhat complicated—different time units roll around to zero at different numbers. Because many applications need such a function, you'd probably move that code out of the per-application code into a framework component if you were manually coding such applications. While you could generate it inline each time it is needed, it seems better to keep the overall code small and conceptually neater by making it into a function; then, when it needs such a function, the code generator needs only to produce a call for it (line 35).