5 Generators

In this chapter we examine the MERL language in which generators are written and the tools for creating and debugging generators.

5.1 GETTING STARTED WITH GENERATORS

In MetaEdit+, generators are used for multiple purposes: code and documentation generation, generator-based text fields in symbols, generator identifiers for non-property types, and to fetch subobjects and their subsymbols for template symbol elements. All these generators are written in the MetaEdit+ Reporting Language, MERL. The code and report generators associated with specific Graph types are edited and managed with the Generator Editor (see Section 5.3) and Debugger tools (see 5.4).

A generator basically does three things: it navigates throught the model structures, extracts information from design elements, and ouputs it to a window or a file. The commands for doing these things form the core of the MERL language. There are also a number of additional commands that enable things like output of diagrams to bitmap files, or execution of external programs.

We will first show how to create (5.1.1) and execute and debug (5.1.2) a simple generator. The primer to MERL in Section 5.2 will bring you up to speed quickly on the approach that MERL takes, and is worth reading even for those familiar with generation and other programming languages. The Generator Editor and Debugger are described in Sections 5.3 and 5.4. The individual MERL commands are explained in detail in Chapter 6, with a quick reference in Section 6.8.

5.1.1 Creating a simple generator

Now it is time to write our first generator. Assuming that we are creating a new generator for the WatchApplication Graph type in our demo repository, we can open the editor by opening the 'Digital Watch' project, selecting 'Simple' (or any other Graph of type WatchApplication) in the **Graphs** list and choosing **Edit Generators** from the popup menu. This will open the Generator Editor as shown below in Figure 5–1:

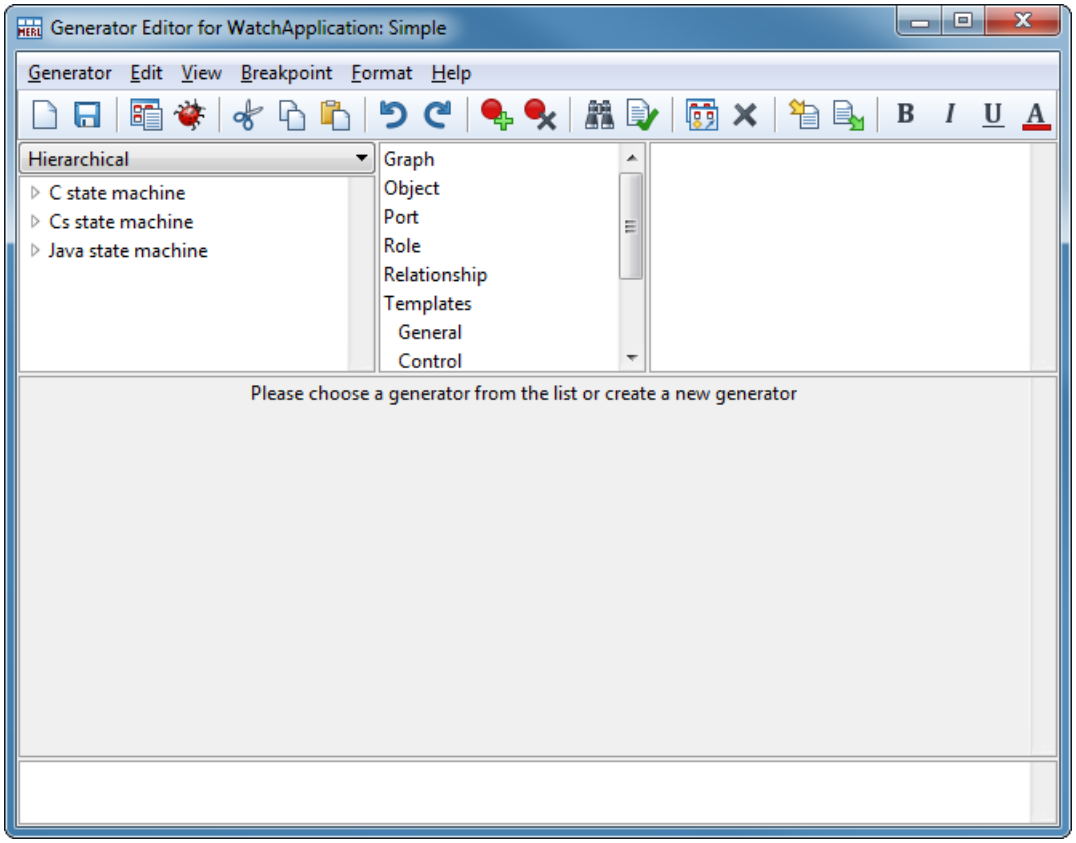

Figure 5–1. Generator Editor.

(If the contents of the window look significantly different, perhaps you opened it on a different Graph type. You can change Graph types with **Generator | Change Graph Type…** or the corresponding toolbar button.)

To create a new empty generator definition, press the **New** button on the toolbar (or **Ctrl+N** or **Generator | New…** from the menu bar), and enter Connections() as a name for the generator when prompted. You will now see the new generator ready for editing at the bottom of the Generator Editor window (as in Figure 5–2).

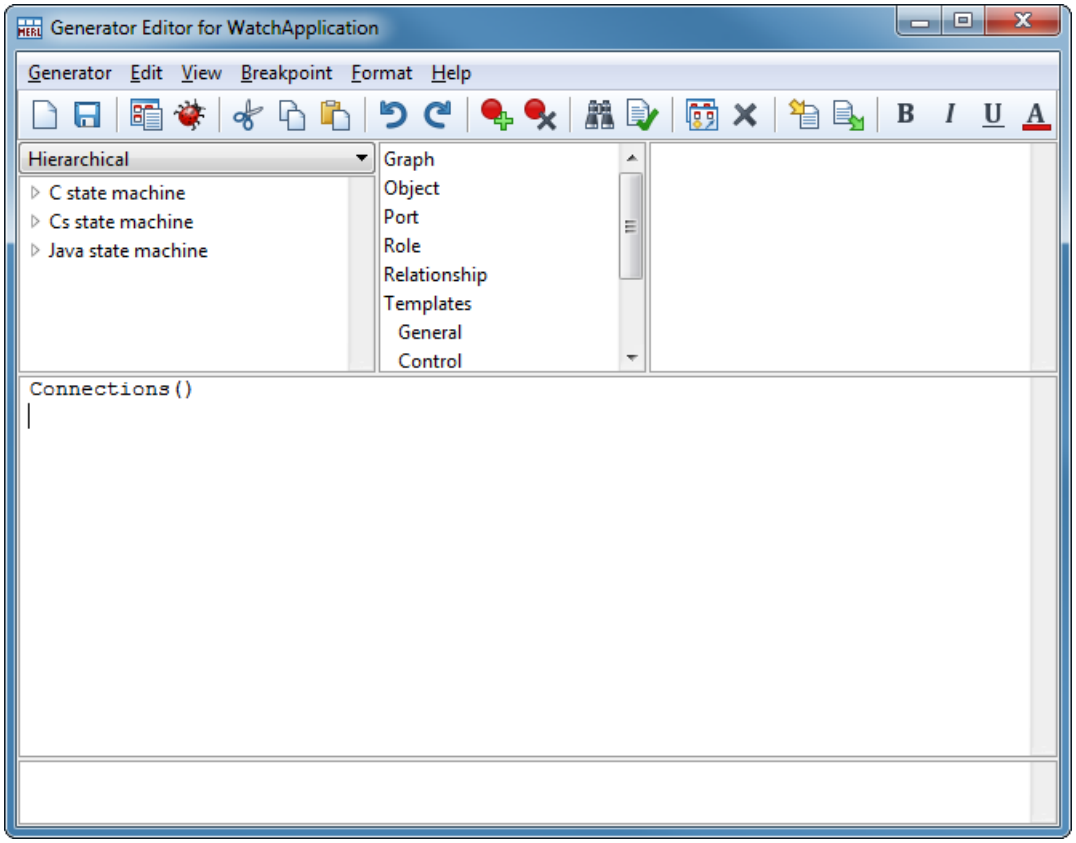

Figure 5–2. Generator Editor with new generator definition.

Let us now enter the following MERL code into the editing area (without the line numbers, they are here just to provide reference points) and press **Save** in the toolbar (or select **Generator | Save** from the menu):

```
01  Connections()
02  foreach .State [Watch]
03  {   'State : '
04      :State name;
05      newline
06      'Connects to: '
07      newline
08      do ~From>()~To.State [Watch]
09      {   '       '
10          :State name;
11          newline
12      }
13      newline
14  }
```

This short piece of code illustrates the basic features of a generator definition. A generator definition accessed from the Generator Editor always has a header at the beginning of the generator definition. There are two ways to construct the header: it can be simply just the generator name followed by (), as above, or use the older `Report … endreport` structure around the generator code (for more information, see Section 6.2.1). The header and possible footer lines, however, are not present in those short generator definitions used in the Symbol Editor or for identifier generators in the Object Tool etc.: since these are not run manually, or called as subgenerators, they do not need a name.

All generators operating on a graph, like our example here, must access their member elements like objects, relationships, roles and ports within a top-level `foreach` loop. In the above example (lines 2 – 14), we loop through the instances of the 'State [Watch]' object type and retrieve specific information from them to output. For each state, we want to output its name, followed by a list of the states it is connected to. After the `foreach` command you will notice the use of the '.' prefix character before the name 'State [Watch]'. The prefix characters '`.>~#:`' make the distinction between Objects, Relationships, Roles, Ports and Properties. For example, in line 2 the '.' preceding the type name tells the generator to fetch objects matching that type name.

In line 3 we enter the loop: lines 3 – 14 will be executed once for each 'State [Watch]' in the graph. Outside the loop we can say that we are 'in' the graph; within the loop we can say that we are 'in' a 'State [Watch]' object.

In lines 3 – 5 we first output the string 'State : ', then the value of the 'State name' property of the current 'State [Watch]' object and then complete the output with a line break. Again, please note the ':' prefix before 'State name' that denotes that a property will follow. Lines 6 – 7 simply output the string 'Connects to: ' followed by a line break.

As you can see, none of these output commands need to specify *where* the output is going: it is going to the current output stream which, by default, just goes to a window, whose result will be shown to the user at the end of the generation. It is also possible to forward the output elsewhere, e.g. to a file or to build up a variable. This will be discussed further in the Primer in Section 5.2.4, and the relevant commands are documented in Section 6.4.1 and 6.5.2.

Lines 8 – 12 contain an important example of a very basic feature of MERL: navigation through bindings. In this case, we want to find out all other 'State [Watch]' objects the current object is connected to, so we use the `do` command to loop through all of them. After the do command you will see the following token:

```
~From>()~To.State [Watch]
```

Though this may look a bit cryptic at first glance, it just says to find all 'From' roles for the current object first, then follow them through the relationship in that binding to the respective 'To' roles and thence to the 'State [Watch]' objects connected to these roles. The '~' prefixes denote the roles and the '>' prefix refers to the relationship. The only extra trick here is that – unlike previously where we used exact type names to access design elements – we now use the special `()` token after '>' to access any kind of relationship that is bound to the 'From' roles.

Within the body of this inner loop we thus iterate over all State objects that are reachable in one From-To 'hop' from the outer loop's State. The content of the inner loop is pretty clear: output some spaces, then the name of the accessed object and a line break. Note how the context has changed within the loop: the current element is now the target State object, whereas in the outer loop it is the source State object. The output of `:State Name;` in line 10 is thus different from the output of `:State Name;` in line 4.

After the inner `do` loop the generator definition is reasonably trivial, just outputting a line break before the main loop is run again for the next State. Note that we are now back in the outer loop, so `:State name;` here would again refer to the same element as in line 4.

Before we proceed with the execution of our example generator, it is worth noting a few things about the MERL keywords and generator names. As a general rule, generator keywords should always be preceded by white space (space, tab, or carriage return), and for backwards

compatibility they can optionally be followed by a semicolon, ';'. Type names that contain spaces, or that are followed on the same line by another command, should be followed by a semicolon. This is not needed in the chaining of type names, where the next type prefix '`.>~#:`' is sufficient to terminate the previous type name, e.g. '`.State [Watch]~To`'.

Generators whose names begin with '_' are hidden from the list shown to the user when choosing **Generate…** elsewhere than the Generator Editor (e.g. from a Diagram Editor). Names like this should thus be used for generators that are intended only for use as sub-generators, e.g. if they assume that the generation output is already going to a file, or that something other than a graph is already in the generator stack. If necessary, a user can still see and even run these generators elsewhere by holding down Shift while choosing **Generate…**.

5.1.2 Executing and debugging generators

Having defined our first generator, it is now time to see it in action. To run the generator, press the **Generate** button in the toolbar (or **Ctrl+R** or **Generator | Generate** from the menu bar). When prompted for the Graph to run the generator on, select 'Simple' (subsequent runs will remember this choice):

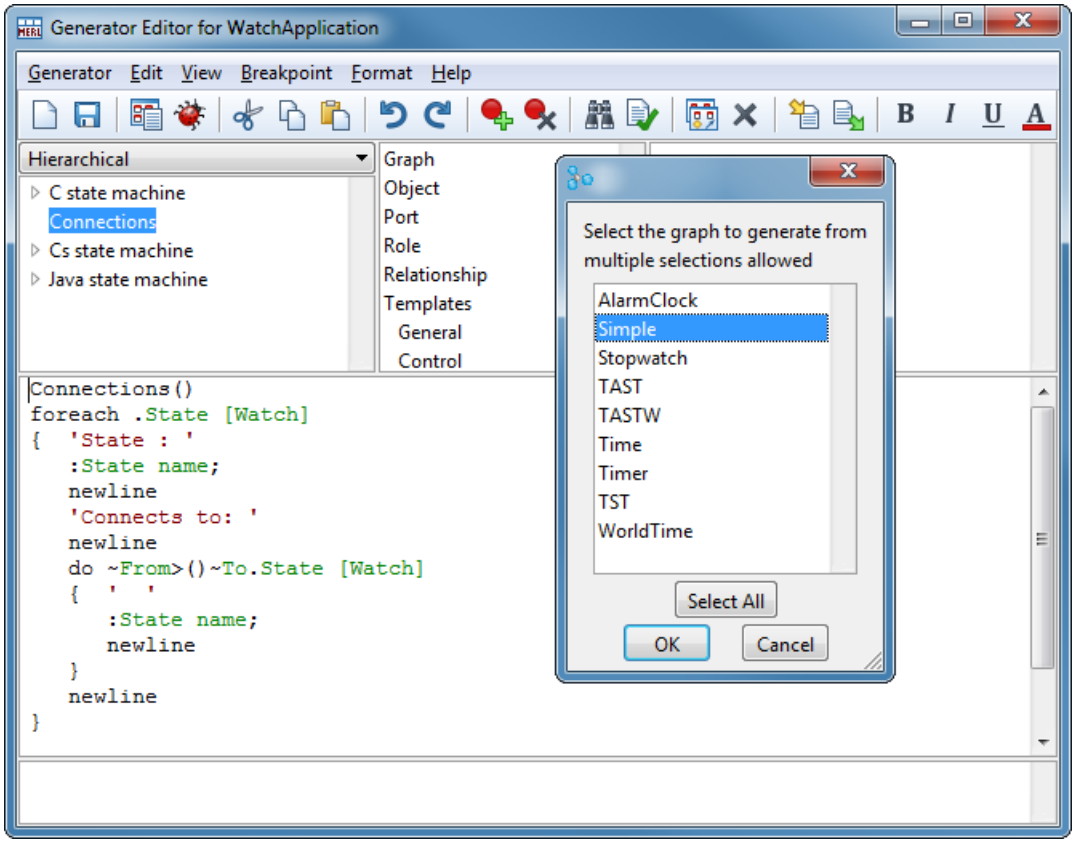

Figure 5–3. Executing a generator.

MetaEdit+ will now execute the generator for the selected graph and open a window with the generator output:

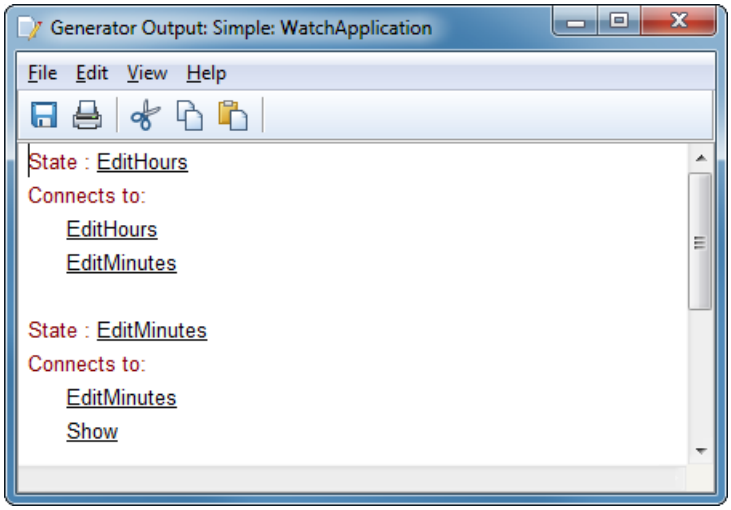

Figure 5–4. Example generator output for 'Simple'.

While it is not usually recommended to manually change the generator output, the Generator Output window nonetheless provides such elementary text editor features as cut/copy/paste, saving and printing. In many cases the Generator Output window is not opened at all, as the generator output is all written to a file; in that case a Generated Files window lists the resulting files, allowing you to look at their contents or open them in their default application.

The Generator Editor also provides an advanced debugging toolset to ease generator development. Tthe detailed documentation for the Generator Debugger can be found in Section 5.4; here we will just look briefly at its basic features. The debugger enables us to trace the execution of the generator line-by-line, set breakpoints at critical parts of our code, and view the current execution context stack as well as the variables and their values.

To debug the execution of a generator, start the generation by pressing the **Debug** button in the toolbar or selecting **Generator | Debug…** from the menu, and select the graph you want to execute the generator for from the list that opens (in our example, we pick 'Simple'). The Generator Debugger will open, loaded with the code of the selected generator as shown in Figure 5–5:

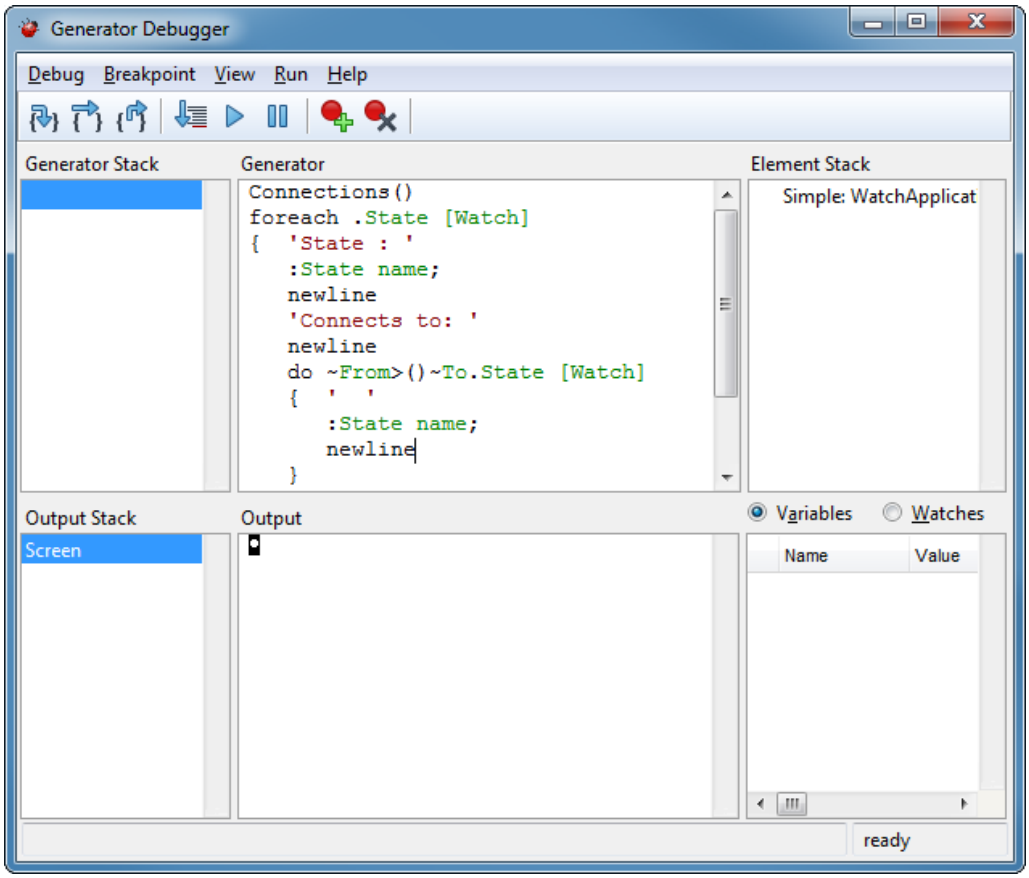

Figure 5–5. Generator Debugger.

The commands for tracing the generation execution are grouped on the left-hand side of the toolbar and in the **Debug** and **Run** menus, while the operations for setting and removing execution breakpoints can be found on the right-hand side of the toolbar or in the **Breakpoint** menu. To try them out, place the cursor at the beginning of the line that says

```
do ~From>()~To.State [Watch]
```

and press the **Add Breakpoint** button in the toolbar (or select **Breakpoint | Add** from the menu). This will place the breakpoint just before the `do` loop. Now, execute the generator up to this breakpoint by pressing **Run** button in the toolbar (or by selecting **Run | Run**). The Generator Debugger will now look like this:

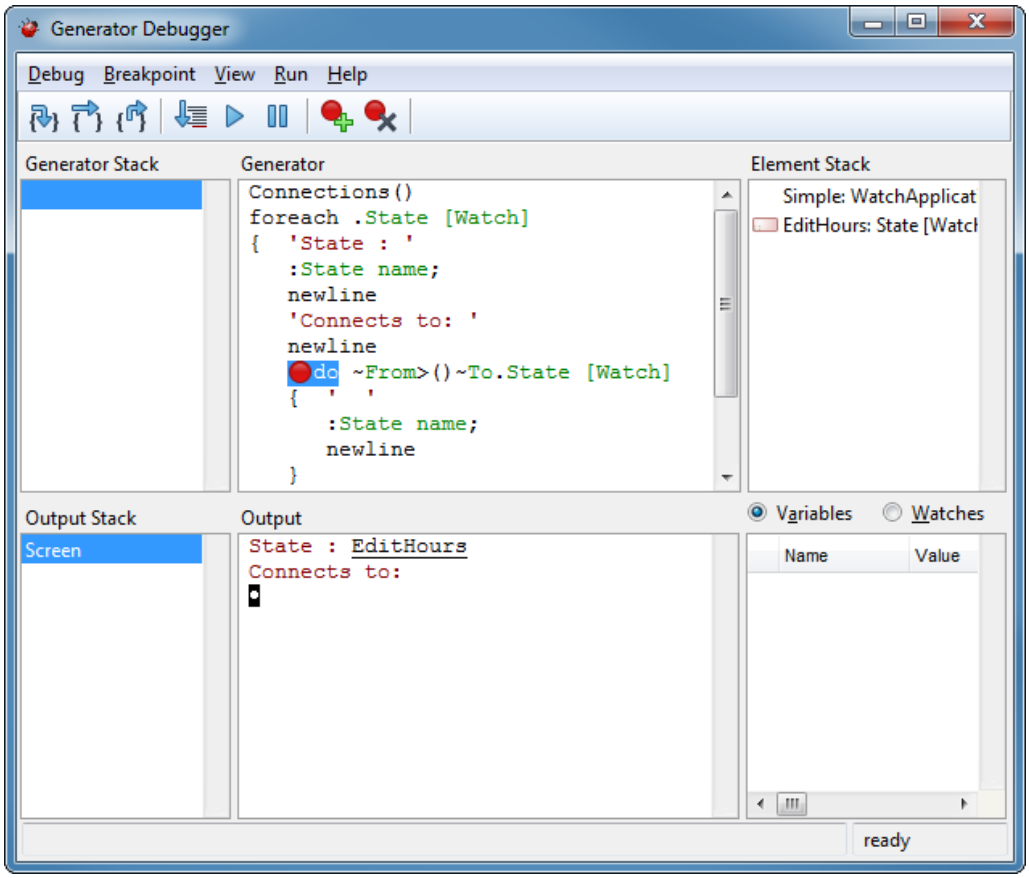

Figure 5–6. Debugging a generator.

As can be seen above, the Debugger Editor now shows an execution snapshot taken at the breakpoint. The next command to be executed is shown with selection highlighting next to the breakpoint in the **Generator** pane, while the various browsers around it provide a view of the execution context and stacks. The generator output produced so far is shown in the **Output** pane. To proceed onwards step-by-step, press the **Step Into** button in the toolbar (or **F5** or **Debug | Step Into** from the menu). While doing this, you will see the **Output** pane and browsers update, as more output is generated and the context changes according to the currently accessed model element. The other options provided for execution tracing are **Step Over** (**F6** or **Debug | Step Over**), **Step Out** (**F7** or **Debug | Step Out**), **Run to Cursor** (**F8** or **Debug | Run to Cursor**), **Run** (F9 or **Run | Run**) and **Break** (**F10** or **Run | Break**).

The Generator Editor and Debugger also provide several ways to use and manage the execution breakpoints. The toolbars on both tools provide simple buttons for adding and removing the toolbars and more advanced operations (like setting condition for the breakpoint or enabling/disabling them) can be found from the **Breakpoint** menu.

5.2 WRITING GENERATORS WITH MERL – A PRIMER

In this primer we will walk through the step-by-step evolution of an example generator. We will use our Watch Example (Figure 5–7) as the source model, and create a generator that will produce specifications sheets for each Watch. An individual specification sheet will provide information about the user interface (number of buttons etc.) and included applications. We will start with a simple textual sheet layout and gradually refine it into an interactive HTML page.

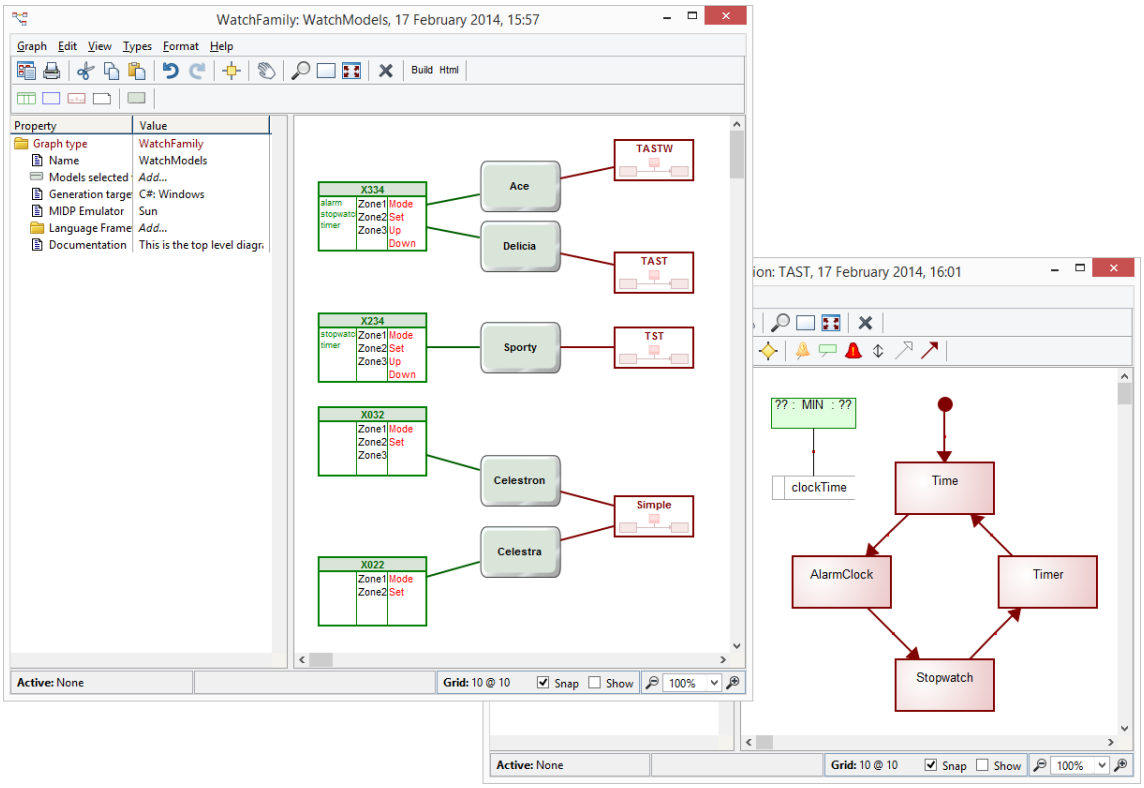

Figure 5–7. The Watch Example, showing the applications in Delicia's LogicalWatch, TAST.

5.2.1 Starting from the beginning

Before writing any actual MERL code for our exercise we have to understand what exactly needs to be done. We have to answer the following questions:

- What design data do we need to extract from the models?
- How do we navigate to and access the data we need?
- Where and how do we output the extracted data?

In our example case, the answer to the first question is simple: we need to get the names of all Watches and the names of their buttons and applications. Studying the models will then reveal the answer to the second question: we have to look at each filled green Watch relationship in turn, getting the Watch's name. We can follow its green role line to its Display object on the

left to get its Buttons' names. We can following its red role line to its LogicalWatch object on the right, and from there down into the LogicalWatch's subgraph to get the names of the applications there.

If we keep the third question simple for the moment and stick with plain text output to the screen, we can already write the first – albeit preliminary – version of our generator:

```
01 SpecSheet()
02 'SpecSheet for ' id newline newline
03 foreach >Watch
04 { 'Watch: ' id newline
05   do .Display
06   { '  Buttons: '
07     do :Buttons { id ' ' }
08     newline
09   }
10   do .LogicalWatch
11   { '  Apps: '
12     do decompositions
13     { foreach .State [Watch] { id ' ' } }
14     newline
15   }
16   newline
17 }
```

When studying the MERL code above it is important to understand the concept of execution context. When the generator is executing, it always operates in the context provided by the currently accessed model element and its environment. This context defines and constrains what design data can be retrieved and which other design elements can be accessed directly at that time. For example, in the context of a Graph, we can always retrieve the property values of the Graph itself and navigate to design elements that are directly in that Graph: objects, relationships, ports and roles. Navigating to an object in the Graph will change this focus – the context is now that of the object. From the context of the object we can retrieve its property values, navigate to roles attached to the object, or navigate to decomposition or explosion subgraphs associated with it.

Armed with this understanding of the execution context, it is now easier to decode our example MERL code. After the obligatory generator header (line 1) we will first print out the heading text for the specification sheet (line 2). The heading text is a combination of a constant piece of text ('SpecSheet for ') and the name of the WatchFamily graph fetched with the `id` command, followed by two newline characters. It is important to note that the `id` command operates on the current context, here the current graph itself – it will return the value of this graph's identifying property.

We continue by looping over all Watch relationships found in the Graph (line 3). The `foreach` loop iterates over each element of the current graph matching the type specified. Inside this loop the context will be that of the currently accessed Watch relationship – therefore the `id` command on line 4 will now fetch the identifier from the Watch relationship. The identifier value will be output with a preceding text label and a newline character.

Lines 5–9 print out the list of button names. To fetch these names, we need to navigate from the current Watch relationship through a Display role to the attached Display object. Navigating within a graph uses a `do` loop, which can specify a chain of navigation steps. In our example, we want to follow the Display role into its Display object, which would look like this:

```
do ~Display.Display
```

We can, however, use a handy shortcut in this case: instead of explicitly stating which role to follow, we can just say which type of Object we want to end up in (as in line 5). Inside the `do` loop (line 6), we are now in the context of the single Display object attached to this Watch relationship. Here, we will first output a header for the list (line 6) and then proceed to retrieve the button names. The buttons are objects contained in a collection property of the Display object, so in order to collect their names we have to loop over that collection using the `do` command (line 7). Within the loop we will then print out the value of the identifying property of the Button at hand, followed by a blank character as a separator. The '`}`' character at the end of line 7 marks the end of this loop, and the `newline` command in line 8 adds a line break after the Button list. Line 9 will then complete the enclosing loop that fetches the Display objects from the Watch relationships.

At this point we are back in the context of our Watch relationship, in the top-level loop. The next step is to collect the names of the Watch's applications.

```
10   do .LogicalWatch
11   { '  Apps: '
12     do decompositions
13     { foreach .State [Watch] { id ' ' } }
14     newline
15   }
16   newline
17 }
```

We start by following the role from the current Watch relationship to its LogicalWatch object (line 10) and printing a header string for the application list (line 11). This time, the content that we want is not in a property of the LogicalWatch, but in a subgraph of it. To follow this link we use the `do decompositions` command (line 12) that will navigate into the associated subgraph. In this subgraph, we will now loop over all State [Watch] objects, which represent the applications, and retrieve their names. As before, to loop over graph elements we use `foreach` (line 13). Within the loop the `id` command will output the name of the State and we will follow it with a blank character as a separator. After that we will close both the innermost `foreach` loop and the enclosing `do decompositions` loop (line 13) and add a line break (line 14) after the list of application names. Line 15 finishes the loop over the LogicalWatches, returning to the context of the Watch relationship.

Line 16 will add an empty line into the output as separator between Watches before the next loop iteration. When all Watch relationships have been iterated over, line 17 finally closes the original `foreach` loop, and our generator ends there.

Executing this MERL code will result in the following output window:

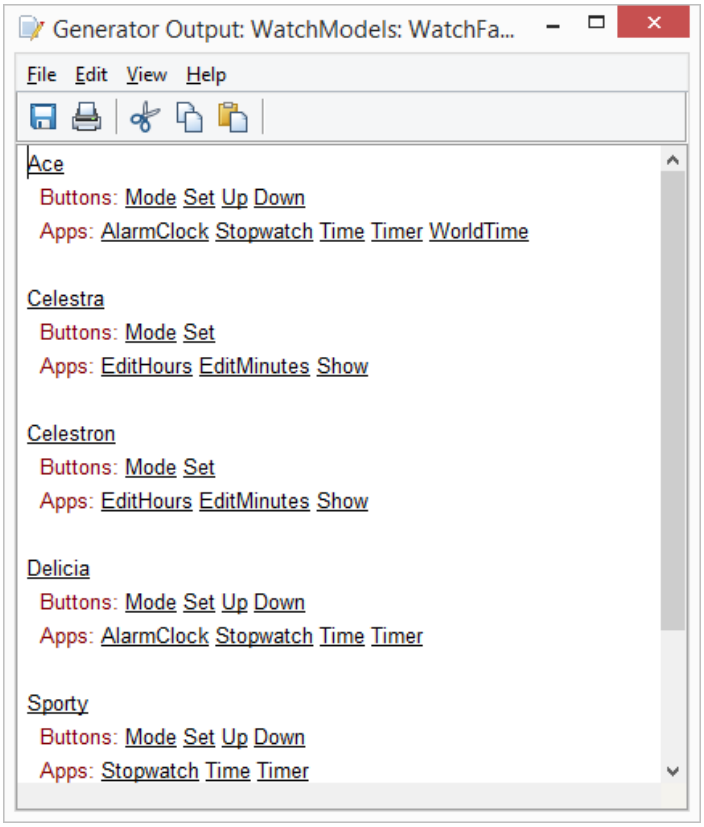

Figure 5–8. Generator ouput for ‘SpecSheet’ generator.

We have also put together a video lesson showing the execution of this example generator, highlighting the navigation and how it changes the execution context. The video can be watched at http://www.metacase.com/webcasts/MERL_Primer_Generator_Context.html.

5.2.2 Context and the element stack

As the generator travels through the `do` and `foreach` loops, it builds up a stack of model elements: each element on the stack corresponds to the context of an enclosing loop, an element we have passed through directly on the way to the current element. In most cases we are only interested in the information found in the current context, but when we need information from earlier parts of our route the stack provides an easy way to access the elements in the context hierarchy. Consider the following fragment from our example code:

```
02 'SpecSheet for ' id newline newline
03 foreach >Watch
04 { 'Watch: ' id newline
```

As explained before, the `id` commands on lines 2 and 4 operate on different contexts: the first one fetches the name of the graph while the second returns the name of the Watch relationship from the current loop iteration. But what if we need to get the name for the graph while we are inside the loop, in the context of the Watch relationship? It is of course possible to use a variable to cache the graph name for future use but MERL also provides a special syntax for cases like this. We can simply refer to the preceding element in the context hierarchy with a `;1` suffix, where 1 is the number of levels back we want to refer to. The level of the current element is 0 and each previous element is removed from it by one, i.e. the previous element in the stack is `;1`, the one before that `;2`, etc. So, to access the name of the graph from within

the `foreach` loop in our example code, we can simply use `id;1`. This applies to property values also, so the same graph name could be fetched using `:Name;1`.

It is also possible to loop over the elements in the stack using a `do` loop. For example, to list all elements of type 'Watch' that we have passed through (starting from and including our current context) would look like this:

```
do stack { if type = 'Watch' then id newline endif }
```

5.2.3 Variables and translators

Having the basic navigation and output in place for our example generator, we can now extend the original requirements by reporting the number of time units provided by each Display into our spec sheet. To do this, we need a count variable, a new loop and some arithmetic. With the modifications our example generator will appear as follows:

```
01 SpecSheet()
02 _translators()
03 'SpecSheet for ' id newline newline
04 foreach >Watch
05 { 'Watch: ' id newline
06   do .Display
07   { '  Number of time units shown: '
08     @count = '0'
09     do :UnitZones { @count++%null }
10     @count newline
11     '  Buttons: '
12     do :Buttons { id ' ' }
13     newline
14   }
15   do .LogicalWatch
16   { '  Apps: '
17     do decompositions
18     { foreach .State [Watch] { id ' ' } }
19     newline
20   }
21   newline
22 }
```

The code that counts and outputs the number of time units can be found in lines 7 – 10. After outputting the header text in line 7, we define a local variable called `count` and initialize it to zero in line 8. Like everything in MERL, this is really just a string rather than a number, but we can interpret it as a number when we want. The `@` prefix means this is a local variable, whose value is visible only throughout this generator; a global variable is prefixed with a `$` and is visible throughout all generators in a given execution run.

In line 9, we loop through the elements in the UnitZone property collection. However, instead of retrieving and outputting any property values we will now just simply increment our counter on each iteration. The Perl-esque command for this:

```
@count++%null
```

takes the `@count` variable, increases it by one and — instead of printing the value of `$count` — throws away the output with the `%null` translator. This output reformatter comes from a set of standard translators that come as part of MetaEdit+'s generator development

environment. In order to be able to use these translators, a subgenerator defining them is called at the beginning of the generator in line 2 (we will see more translators in action shortly). After all the iterations the variable `@count` will then hold the number of available time unit slots. This number will then be output (followed by a `newline` command) in line 10.

Having completed the basic functionality of our example generator, we will now see how to refine it further, starting with outputting to a file.

5.2.4 Streams and files

All generator output is appended to the current output stream. By default, the stream just goes to a window, whose result will be shown to the user at the end of the generation. It is also possible to direct the output to files (see Section 6.4) or variables (Section 6.5). Temporary streams are also used by some commands to form their arguments, as we will see shortly.

In our example case, we can simply forward the output to a file by using the `filename…write…close` command:

```
01 SpecSheet()
02 _translators()
03 filename id%file '.txt' write
04   foreach >Watch
05   {
 .
 .
 .
22   }
23 close
```

Everything between `filename` and `write` goes to make up the name of the file; everything between `write` and `close` is written to that file. The name of the file here in line 3 is thus the value of Graph's identifier property (i.e. 'WatchModels'), translated to be suitable for a filename with the `%file` translator, and followed by '.txt'. After the `write` command, all output will be collected to go into this file until the `close` command at the end of the generator (in line 23), which closes the file stream and actually performs the write to disk.

The example above collects the spec sheets for all Watch models into one file. In order to have a separate file for each model, we can simply move the `filename…write…close` command inside the main `foreach` loop:

```
01 SpecSheet()
02 _translators()
03 foreach >Watch
04 { filename id%file '.txt' write

 .
 .
 .
22   close
23 }
```

Instead of one 'WatchModels.txt' file we will now get 'Ace.txt', 'Delicia.txt, 'Sporty.txt' etc. All files are generated into the default directory but we can also prefix the filename with

directory information, if needed (the `sep` command outputs the platform's directory separator, '\' or '/'):

```
filename 'reports' sep id%file '.txt' write
```

MetaEdit+ also provides a pre-defined subgenerator that outputs the default relative path for the reports directory created by the installer. Using this subgenerator, the above piece of code would look like this:

```
filename
  subreport '_default directory' run
  id%file '.txt'
write
```

The filename can thus be constructed using any MERL commands between `filename` and `write`: all their output goes to build up the filename. You can also override the existing '_default directory' subgenerator or write a new one that suits your purposes. Alternatively you can store a directory into a variable for further reference:

```
$myDir = 'reports\'
filename $myDir id%file '.txt' write
```

5.2.5 Navigation revisited

In the initial version of our example generator we use e `foreach` and `do` loops, without any filtering or explicit ordering, to go through the elements on the Graph level, to access property values and subgraph links, and to navigate along roles and relationships. While this approach seems to work fairly well in our limited example, it is not perfect by any means. For example, the outermost `foreach` loop handles the WatchModel relationships in the default alphabetical order, which may not be the order we want them to appear in our spec sheet. If we want them to appear in the same order as they appear vertically in our diagram, we can augment the `foreach` command with an `orderby` clause:

```
foreach >Watch; orderby y num
```

The above can be read simply as 'go through all the Watch relationships in ascending order according to their y coordinates, sorted as numbers'. The `num` is important: otherwise they will be sorted as strings, which places '100' before '99'. Note also the semicolon to show MERL where the relationship type name finishes; otherwise the rest of the line would be considered part of the type name. (A line break or open brace '{' would also suffice to end the type name, since '{' cannot be in a literal type name in MERL unless escaped as '\{'.)

Another problem arises when we are outputting the button names. We used a blank character to separate the names, but what would happen if we were using the more usual comma character as a separator? The MERL code would look like this:

```
do :Buttons { id ', ' }
```

The example output would now be:

```
Buttons: Mode, Set, Up, Down,
```

This isn't ideal, as we don't want an extra comma after the list of the names. The simple solution is to replace `do` with `dowhile`: when iterating over the last element of the loop, `dowhile` skips all simple string commands at the end of the loop:

```
dowhile :Buttons { id ', ' }
```

The output will now look like this:

```
Buttons: Mode, Set, Up, Down
```

There is also another ordering problem looming when we are fetching the names for applications within the LogicalWatch's subgraph. The loop command we are using for this retrieves the State [Watch] objects in the default alphabetical order. Adding an `orderby` clause doesn't help here, as the order we want isn't easily expressible as a sortable value for each element. The best order is probably that of the Transition relationships, which define the cycle of applications within the LogicalWatch. The cycle starts from the start state and flows from each State [Watch] object to the next through the directional Transition relationships. This requires somewhat more complex navigation in the subgraph. As there is only one Start in each WatchApplication graph (as defined by an occurrence constraint), we can use a simple `foreach` command to get to the starting point for our navigation. The MERL code for this and the consequent navigation to the first State [Watch] object looks like this:

```
foreach .Start [Watch]
{ do ~From~To.() { /* some code here */ } }
```

We find the Start, then navigate along the From role and To role into the object at the other end. (Here we use the wildcard `.()`, meaning any object; we could also explicitly write `.State [Watch]`.) The navigation from one State [Watch] to the next is similar:

```
do ~From~To.()
```

There is now, however, a catch: as there is no explicit Stop state in our graph and the number of State [Watch] objects differs between different graphs, how do we know when to stop looping? The solution is to keep track of the objects as we pass through them, and stop when we encounter an object we have already visited. The next section discusses how this can be done with recursion and subgenerators.

5.2.6 Modularization and recursion

In order to go through all State [Watch] objects by their Transition relationships, until we reach an object we have already seen, we need to store the information about the objects we have visited and check each new object against that list. If the object can be found from the list, we know that we have completed the full cycle and can stop. So what we need to do is:

1) Initialize a variable to store the list of objects we have visited.
2) Fetch the first State [Watch]
3) Check we have not already visited the current State [Watch]
4) Output the name of the current State [Watch]
5) Add the current State [Watch] to the variable
6) Navigate to the next State [Watch] in our cycle
7) Go back to 3)

To implement the above in MERL we need a subgenerator that carries out steps 3–7, calling itself at the end to implement the recursion. A subgenerator is just a normal generator

definition, executed from within another generator. The subgenerator is executed in the same context as the parent generator is when the call is made. It thus has the same element stack as the parent, the same stack of output streams, and the same global variables (but its own new set of local variables). See the definition of the `subreport` command in the Section 6.3.11 for more details and ways of using subgenerators.

To keep track of the visited objects, we need a unique identifier for each. The name might work in some cases, but more reliable is to use the unique identifiers produced by the oid command, whose output looks like `3_123` and is guaranteed unique within a repository. We can simply append the oid of each object to a string variable, with a space as a separator: `' 3_123 3_129 3_158 '`. This lets us see if an object is in the list with a simple =~ wildcard comparison, e.g. `'* 3_158 *'` matches but a new `'* 3_222 *'` does not. Having the spaces saves us from mistakenly matching `'*3_222*'` with `'3_2221'`.

The subgenerator for this case can thus be:

```
01 _handleStateAndRecurse()
02 if not $visited =~ oid%wildsp then
03   if $visited <> ' ' then
04     ', '
05   endif
06   id
07   variable 'visited' append oid ' ' close
08   do ~From~To.()
09   { _handleStateAndRecurse() }
10 endif
```

The `if` clause around the whole generator in line 2 checks that the current State [Watch] object has not already been visited (note the use of the `%wildsp` translator, which adds the spaces and wildcard asterisks around the oid). If we are good to go, lines 3 – 6 will output the name of the current element. As we want the names to appear as a comma-separated list and without any additional commas at the end of the list, we use a little trick in lines 3 – 5: we check if we have already visited any objects and if we have, we will output `', '` before the actual name. This means that all but the first name in the list will be preceded by a comma and a space. After outputting the name, line 7 appends the current object's unique identifier and a space to the list of visited objects, and lines 8 – 9 take care of the next iteration by calling this same subgenerator recursively.

→ *As the _handleStateAndRecurse subgenerator operates in the context of a WatchApplication graph type, we want to define it for that graph type, not for the WatchFamily graph type that the SpecSheet generator was defined for. You can open another Generator Editor, or change the Graph type of the current Editor with Generator | Change Graph Type…. This kind of division of responsibility to different types is typical for object-oriented programming and the object-oriented principles also work in cases where a generator is applicable to more than one graph type: there is no need to duplicate it to all graph types as it can be defined in a common supertype, or in the supertype of all graph types, Graph itself, which makes it available for all of them.*

With all the aforementioned changes, our SpecSheet generator will now look like this:

```
01 SpecSheet()
02 _translators()
03 foreach >Watch; orderby y num
04 { filename
05     subreport '_default directory' run
06     id%file '.txt'
07   write
08     'Watch: ' id newline
09     do .Display
10     { '  Number of time units shown: '
11       @count = '0'
12       do :UnitZones { @count++%null }
13       @count newline
14       '  Buttons: '
15       dowhile :Buttons { id ' ' }
16       newline
17     }
18     do .LogicalWatch
19     { '  Apps: '
20       do decompositions
21       { foreach .Start [Watch]
22         { $visited = ' '
23           do ~From~To.()
24           { _handleStateAndRecurse() }
25         }
26       }
27       newline
28     }
29     newline
30   close
31 }
```

The use of the recursive subgenerator requires one addition to our SpecSheet() MERL code. In line 22 we initialize the `$visited` variable. It is a global variable that stores the list of visited objects and is accessed and updated in the _handleStateAndRecurse subgenerator. After these changes, the generator output will now look like this:

```
Ace
  Number of time units shown: 3
  Buttons: Mode, Set, Up, Down
  Apps: Time, AlarmClock, Stopwatch, Timer, WorldTime
```

So far our target output has been plain text without any real layout. In the next chapter we will discuss how to format the output for a specific purpose.

5.2.7 Formatting output for a specific purpose

Our output thus far has been plain text without any serious structure or layout. Here, as the next task for refining our example generator, we will spice up our spec sheet with some HTML formatting. The outcome of this exercise looks like this:

```
SpecSheet()
_translators()
@defaultDir = __(subreport '_default directory' run)
@cssFilename = _myFilename('css')
filename @defaultDir @cssFilename write _css() close
foreach >Watch; orderby y num
{ filename @defaultDir _myFilename('html') write
    '<html>' newline
    '<link rel="stylesheet" type="text/css" '
      'href="' @cssFilename '"/>' newline
    '<body>' newline
    '<h1>' id '</h1>' newline
    '<p class="description">' :Documentation '</p>'
    newline
    do .Display
    { '<p class="info">Number of time units shown: '
      @count = '0'
      do :UnitZones { @count++%null }
      @count '</p>' newline
      '<p class="info">Buttons: '
      doWhile :Buttons { id ' ' }
      '</p>' newline
    }
    do .LogicalWatch
    { '<p class="info">Apps: '
      do decompositions
      { foreach .Start [Watch]
        { $visited = ' '
          do ~From~To.()
          { _handleStateAndRecurse() }
        }
      }
      '</p>' newline
    }
    '</body>' newline
    '</html>' newline
  close
}
```

As can be seen above, the HTML tags have been embedded as part of the output stream. We have followed good HTML practices by sticking with the basic HMTL tags within the generator itself and having a separate style sheet for defining the actually look and layout. The style sheet definition is encapsulated into a subgenerator called _css() and written to a CSS file that is then referred from the HTML documents created for the Watch models. This is a good way to incorporate predefined and constant piece of code or some other text into your generator structure. Having the constant parts hidden in their own subgenerators doesn't obscure the core part of your generator with excessive content, and leaves that fixed content easy to access and modify if needed.

You can initially define _css() as a new empty generator; later you can make its contents the CSS you want, embedded in a MERL string. Since any single quotes inside MERL strings must be escaped by doubling them, there is a menu item **Edit | Paste as Quoted** that will allow you to copy the literal CSS from elsewhere, and have it pasted into MERL as a string surrounded by single quotes, and with contained single quotes doubled.

We also refer to a _myFilename generator, which takes the desired file extension as an argument, and outputs a filename with that extension based on the current element's name. You can define it like this:

```
_myFilename(@ext)
   id%file '.' @ext
```

5.2.8 Additional exercise

As a final task of our exercise we will add some interactivity into our HTML documents by linking each Watch's page to the spec sheets of previous and next Watches. There is again a little catch here: the first page only has a link to the previous page and the last page only has a link to the previous page, while intermediate pages can link both ways. This means that there is going to be some variation when outputting the HTML for individual Watch pages.

There is of course more than one way to implement this requirement. We will explore two different ways to do this, starting with the one that is probably the most obvious to most people with any previous programming experience. In this solution we create an index for each Watch model first and then use this index when we need to fetch the information about the previous and next models while creating the spec sheet for a Watch model. The relevant parts of this solution look like this:

```
@ix='0'
foreach >Watch; orderby y num
{ local 'ix_' @++ix write id close }

@ix='0'
foreach >Watch; orderby y num
{ .
  .
  '<p class="navigation">'
  @prev = __(local 'ix_' @ix++ read)
  @next = __(local 'ix_' @++ix-- read)
  if @prev then
    'Previous model: ' _linkHTML(@prev)
  endif
  if @prev AND @next then ' | ' endif
  if @next then
    'Next model: ' _linkHTML(@next)
  endif
  '</p>' newline
  .
  .
}
```

After executing the first three lines we will have the MERL equivalent of an array, a set of local variables that store the name for the Watch model at the respective index, i.e.:

```
@ix_1 = Ace
@ix_2 = Delicia
@ix_3 = Sporty
@ix_4 = Celestron
@ix_5 = Celestra
```

After this initialization, we reset @ix and enter the main `foreach` loop where we assign the @prev and @next local variables with the names of the Watch models preceding and following the one currently at hand. As the syntax here may initially appear a bit cryptic, we will walk through it step-by-step. Consider the first assignment:

```
@prev = __(local 'ix_' @ix++ read)
```

Starting from within the parentheses, we first construct the name of the index variable pointing to the previous Watch model:

```
'ix_' @ix++
```

The name of the variable starts always with prefix 'ix_' that is followed by the index number. The counter arithmetic in this case reads as: output the current value of `@ix` first, and then increment @ix by 1. Hence, we will get the name for the index variable (i.e. 'ix_0', 'ix_1' etc.) and have our counter updated. After this step, the assignment line is thus effectively:

```
@prev = __(local 'ix_0' read)
```

The `local…read` command will fetch the value for the named local variable. At this point it is worth pointing out that there is no ix_0 among our existing index variables. Reading the value of an as-yet-undefined variable just returns an empty string, which is what we want here as there is no predecessor for our first Watch model. The surrounding __() is just used so that we can have multiple or non-trivial commands on the right-hand side of the assignment: it's actually a call to a subgenerator called '__', which simply returns the value of the argument built up between the parentheses. The subsequent iterations will result in the values of variables @ix_1, @ix_2 etc., which will give us the name of the preceding Watch to store in @prev. We can then use that later to output the appropriate HTML for a link to the file of the same name, in a subgenerator _linkHTML(@prev) whose definition is left to the reader.

The assignment of the @next variable is otherwise similar to above, but the name for the index variable is constructed differently:

```
'ix_' @++ix--
```

The name starts with the 'ix_' prefix again but is followed now by the index number of the next Watch model. The counter arithmetic thus now reads: increment counter by 1 (to get the next index), output its value, and then decrease it by one (to get back to current index as we need it during the next step of iteration). Otherwise the assignment executes as with @prev. On the final Watch model we will encounter the variable name 'ix_6' that is not pointing anywhere and will end up having @next as blank – which is again what we want as there is no Watch model following the last one.

The solution above uses MERL's associative variables like an Array, i.e. it forms key and value pairs where the key is a numerical index and the value is a Watch model name. However, this requires us to keep track of index counters during the creation and use of the array. It is also possible to use associative variables in a more organic way, more like a Dictionary or HashMap than an Array, by using Watch names as keys instead of building numerical indices to use as a key. This way we no longer need to maintain the information about indices. The new variable names also make the code more readable and understandable. Rather than trying to write "C in MERL", we are using MERL in a more natural way. Using this approach, the complete SpecSheet() generator will now look like this:

```
SpecSheet()
_translators()
@defaultDir = __(subreport '_default directory' run)
@cssFilename = _myFilename('css')
filename @defaultDir @cssFilename write _css() close
@prev = ''
foreach >Watch; orderby y num
{     local 'prev_' id write @prev close
      local 'next_' @prev write id close
      @prev = id
}
foreach >Watch; orderby y num
{ filename @defaultDir _myFilename('html') write
    '<html>' newline
    '<head><link rel="stylesheet" type="text/css" '
      'href="' @cssFilename '"/></head>' newline
    '<body>' newline
    '<h1>' id '</h1>' newline
    '<p class="doc">' :Documentation '</p>' newline
    do .Display
    { @count = '0'
      do :UnitZones { @count++%null }
      '<p class="info">Number of time units shown: '
      @count '</p>' newline
      '<p class="info">Buttons: '
      dowhile :Buttons { id ' ' }
      '</p>' newline
    }
    do .LogicalWatch
    { '<p class="info">Apps: '
      do decompositions
      { foreach .Start [Watch]
        { $visited = ' '
          do ~From~To.()
          { _handleStateAndRecurse() }
        }
      }
      '</p>' newline
    }
    '<p class="navigation">'
    @prev = __(local 'prev_' id read)
    @next = __(local 'next_' id read)
    if @prev then
      'Previous model: ' _linkHTML(@prev)
    endif
    if @prev AND @next then ' | ' endif
    if @next then
      'Next model: ' _linkHTML(@next)
    endif
    '</p>' newline
    '</body>' newline
    '</html>' newline
  close
}
```

The execution of the first `foreach` loop will create the following variables:

```
@prev_Ace =
@next_Ace = Delicia
@prev_Delicia = Ace
@next_Delicia = Sporty
@prev_Sporty = Delicia
@next_Sporty = Celestron
@prev_Celestron = Sporty
@next_Celestron = Celestra
@prev_Celestra = Celestron
@next_Celestra =
```

Now, by knowing the id (i.e. the name) of the current Watch model we can construct the names for variables that point to its processor and successor. Within the second loop we will now use this trick to create the @prev and @next local variables as we did earlier. Otherwise the generator remains the same as before.

The final output of this generator will now result in a web page like this:

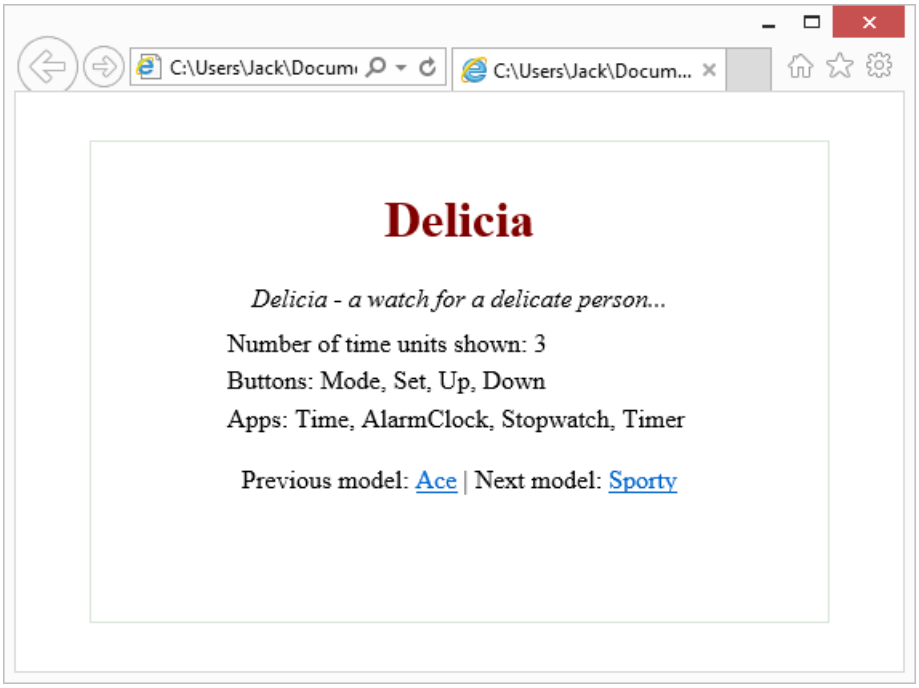

Figure 5–9. Interactive web page generated from the Watch Example.

The generator could still be improved, e.g. to make sure that any characters illegal as literals in HTML, like & and <, are escaped when found in the model. This can be handled simply by appending %xml to the relevant output commands, e.g. id%xml and :Documentation%xml.

Adding these final touches to our exercise completes this primer for developing generators with MERL in MetaEdit+. After this general overview of generator development practices, the following chapters will provide more comprehensive descriptions of the MERL development tools like the Generator Editor and Debugger, and of the individual MERL commands.