### CHAPTER 7

## HOME AUTOMATION

In this chapter, we will look at an example of a Domain-Specific Modeling (DSM) solution for an embedded system that works closely with low-level hardware concepts. While the DSM solution itself succeeded in its aims, this case encountered a number of difficulties. Rather than only providing examples of the triumphal march of DSM, we hope that looking honestly at these problems will prove useful in helping you to avoid them. Names and minor details have been changed to protect the innocent.

7.1 INTRODUCTION AND OBJECTIVES

The company, which we will call Domatic, worked as a co-manufacturer and solution provider, producing a variety of hardware and software products. The focus was on M2M, Machine-to-Machine communication, applied to domains including energy, home automation, telecommunications, and transport.

Domatic wanted to investigate DSM, to see if and where it could be applied in their work. They were looking for higher levels of productivity through automating parts of the production of software and configuration information. As Domatic had no experience of DSM, and indeed little of any kind of modeling, they engaged a consultant from a company experienced in DSM to help them perform a proof of concept. As an example domain they chose an existing home automation system. Although there were no plans to build a large range of new variant systems in that domain, a few variants already existed, so it seemed a good candidate domain for DSM.

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

140

### INTRODUCTION AND OBJECTIVES

7.1.1 Target Environment and Platform

The home automation system chosen for the proof of concept offered control of a range of devices including heating, air conditioning, lights, and security. The focus for the proof of concept was on a telecom module, which allowed remote control of the system over a phone line. In addition to remote control, the telecom module also allowed the remote update of its software, and commands from the main home automation system to dial out to a remote number to report alarms or log other data. The module had already been designed and built, and a few variants of it had been made as part of products for different clients.

The telecom module was operated remotely by a normal call from a phone. The module used voice menus to provide information and offer the user choices, which he could activate by pressing buttons on the phone keypad. The module used a standard telecom chipset to recognize the frequencies of the DTMF tones and translate them back into the simpler form of which button had been pressed.

The voice menus used real speech, sampled and stored in the module. As this was an embedded device, the speech was broken down into reusable sampled units of words or phrases to save memory. An actual sentence was played back as a sequence of these samples.

Clients supplied a sketch of the desired voice menu, for example in simple flow charts. These were fleshed out by Domatic into a spreadsheet format which added the technical details. For instance, sentences were broken down into sample units, and the choices were implemented as jumps to another row in the spreadsheet. Each row of the spreadsheet represented a certain memory address containing one primitive command: play a certain voice sample, jump to a certain memory address, assign a value to a register, and so on. Listing 7.1 shows the spreadsheet for a loop that reads out all five modes in the system, and tells the user which button to press for each.

Listing 7.1 Spreadsheet to read out the list of modes.

**Address Command Argument**

00A1 Load A 00

00A3 Add A 01

00A5 Say 'For'

00AE SayMode A

00AF Say 'press'

00B8 SayNumber A

00B9 Test A < 05

00BB IfNot 00BC Jump 00A3

As the listing shows, the spreadsheet forms an assembly language program. An in-house assembler processed the spreadsheet into a binary file that implemented the program, running on an 8-bit microprocessor. As opposed to third-generation programming languages such as C or Java, assembly languages are specific to a given microprocessor, and sometimes also to a lesser extent to a given domain of use. This in-house assembly language included a variety of “Say” commands,

which would play a sample. Most samples were specified simply by memory address index and length: the actual samples were burned to an EEPROM. For some frequently used samples, a specific shorter command could be used, for example, “SayNumber B” to play the sample corresponding to the value of register B: “one” for 1 and so on.

7.1.2 DSM Objectives

Unlike other cases, there were no clear objectives for a DSM solution in this domain. The main goal was to use this example to examine the applicability of DSM in low-level embedded software development in general. As Domatic produced solutions based on other companies’ requests, the actual domains varied with each new customer. An important goal was therefore the ability to quickly create a new DSM solution, including the modeling language, generator, and tool support.

Domatic used no specific method for software development. Their developers would sometimes draw simple flow charts or state diagrams, either before or after they wrote the code. Reuse of code from older projects followed the “industry standard” practice of simply copying whole code files and changing parts. Recognizing the problems inherent in this approach, Domatic hoped that DSM solutions would increase the consistency of their software development and the reusability of designs and code.

As their current development relied largely on ad hoc or post hoc documentation and testing, Domatic were also interested in the fact that DSM models were at a high enough level of abstraction to serve as a communication medium with clients. The models could serve as the formal requirements specification, and at the same time as internal design documentation. Through code generation, the models could also be immediately tested.

7.2 DEVELOPMENT PROCESS

The DSM solution was developed in MetaEdit+ 3.0 by a consultant from MetaCase and an expert developer from Domatic. The consultant supplied the DSM know-how and actually built the metamodel and generators. Domatic supplied the understanding of the domain and the required code, and also made an extension to their spreadsheet assembler. The development of the DSM solution set out to follow the process for a proof of concept described in Section 13.3. As we shall see, however, not all went according to plan.

7.2.1 Before the Workshop and Day 1

Domatic had supplied the consultant with material about their domain and language three weeks before the workshop. The material covered the whole home automation domain, focusing on the telecom module. A week before the workshop they

### DEVELOPMENT PROCESS

emphasized a particular description of the whole home automation system and how it interacted with its sensors, actuators, keypad, screen, data modem, and DTMF voice control.

The first day of the workshop was spent building up a shared picture of this wider domain, resulting in a modeling language containing concepts like sensors and actuators. By the end of the first day, it was apparent that this languagewas too generic. Just knowing that there is a sensor called “smoke detector” connected to the system, and an actuator called “fire alarm”, is not enough to generate meaningful code. The language would thus be useful for describing whole systems, and possibly for configuration, but not for demonstrating DSM with full code generation.

7.2.2 Day 2: If at First You Don’t Succeed...

The second day of the workshop had the hard deadline of a meeting at 1 p.m. to present the results to management. The first part of the morning was spent establishing the area of the domain to be covered. Rather than focusing on the boundary, which is hard to lay down precisely at an early stage, the consultant and Domatic experts identified the central concepts of the domain. The modeling language had to be able to specify a section of voice output built up from text fragments, and a choice based on DTMF input. A small modeling language for this was built in MetaEdit+ in 25 minutes, 10:40–11:05.

Using this VoiceMenu modeling language, Domatic built a small example model and sketched the corresponding code. As both the modeling language and the assembly language were specific to the same narrow domain of Domatic’s telecom module, there was a good correspondence between model elements and lines or blocks of code. The consultant could thus build a basic code generator for the skeleton modeling language in 10–15 minutes.

In the remaining time up to 11:40, the modeling language was extended to handle more than the core cases: what to dowhen the user did not follow the voice instructions or when the voice output varied according to the state of the system. Concepts and control paths were added for timeouts and invalid input in DTMF, and for system calls to manipulate and test registers. Because of time constraints, the system calls were left free-form: the modeler had to know to use one of several possible assembler commands.

From 11:40 to around 12:00, the code generator was extended to handle the normal usage of these new concepts. The basic rules for how elements could be connected in the models were already specified along with the concepts, but there was no time for even the more obvious finer rules or checks. With an hour left until the meeting, there was time to finalize the example model, eat a hurried lunch, and prepare slides for the presentation to management.

7.2.3 Further Development

After the meeting, there was a little time left to refactor the DSM solution. The modeling language was split into two diagram types, with the top level showing the voice menu and DTMF interactions. Each voice element there exploded to a lower-level diagram showing how it was built from static text fragments, varied by system calls. After the proof of concept workshop, the consultant finished this refactoring. He also added the missing parts of the code generator based on the sample code provided and sent the results to Domatic. These additions after the workshop took at total of two hours.

7.3 HOME AUTOMATION MODELING LANGUAGE

This particular DSM solution was never taken into wider use at Domatic, and thus has not experienced the normal evolution and rounding off of sharp corners. Some minor adjustments have been made to the version presented here to remove proprietary details. In this section, we will look at the modeling language and its metamodel; readers who prefer to see practice before theory can first take a quick look at the example models in Section 7.4.1.

7.3.1 Modeling Concepts and Rules

There were two modeling languages making up this DSM solution. The VoiceMenu language described the high-level interaction from the point of view of the caller. This language was thus useful not only for specifying the hierarchical structure of the voice menus, but also for discussing this structure with the client, or for providing documentation to the end users. Each part of the model that specified speech or system actions was further detailed in the lower-level VoiceOutput language. This language took the place of the earlier assembly language statements written in the spreadsheet.

Figure 7.1 shows the definition of the VoiceMenu modeling language. The main concepts are the VoiceOutput, where the telecom module says something to the caller, and the DTMF\_Input, where the module waits for the caller to press touch tone buttons to make a choice. The normal flow is from a singleton Start object to a VoiceOutput, which gives instructions about possible choices, to a DTMF\_Input that waits for input from the caller. The type of input expected is specified in a property of the DTMF\_Input object as either “character” or “string” (for simplicity, we will concentrate here on single character input). For each possible input there is a ConditionalFlow relationship to another VoiceOutput. Mostly, this will be a test for equality with a given character specified in the ConditionalFlow, but slightly more complex conditions like >= could also be specified.

If an invalid key is pressed there is an InvalidInput flow, normally back to the previous VoiceOutput, that is, the instructions for this choice. For cases where no input is received there is a Timeout flow, which specifies how long to wait before it is followed, again normally back to the previous VoiceOutput. AVoiceOutput can also be directly followed by another VoiceOutput, allowing reuse at this level.

In addition to the rules specified here about how objects can be connected with various flows, there are also some more specific constraints. As usual, a Start object can be in just one From role, to prevent ambiguity in the initial control flow. In a

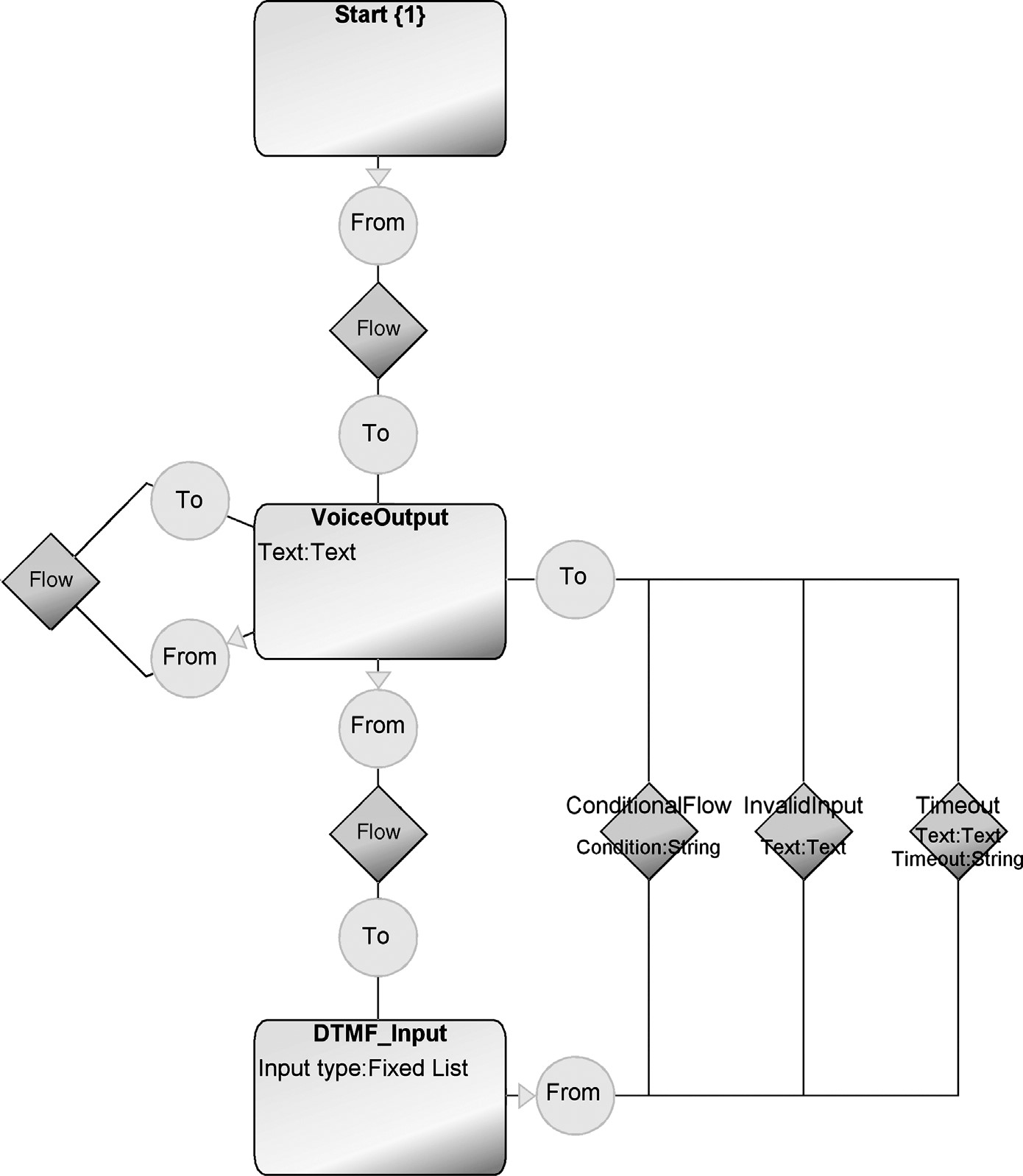


FIGURE 7.1 Top-level metamodel, VoiceMenu

similar way, a DTMF\_Input object may only be in one InvalidInput and one Timeout relationship: there would be no way to choose between several. There will however normally be several ConditionalFlow relationships, as they each specify their own Conditions: the various keys that can be pressed.

Each VoiceOutput object, InvalidInput relationship, and Timeout relationship specifies in a lower-level diagram the actual speech it produces: the text property in the elements themselves is a description for the convenience of the modeler. The structure of explosions from the top-level VoiceMenu to lower-level VoiceOutput diagrams is shown in Fig. 7.2. Most often, the speech used for all InvalidInputs will be the same,

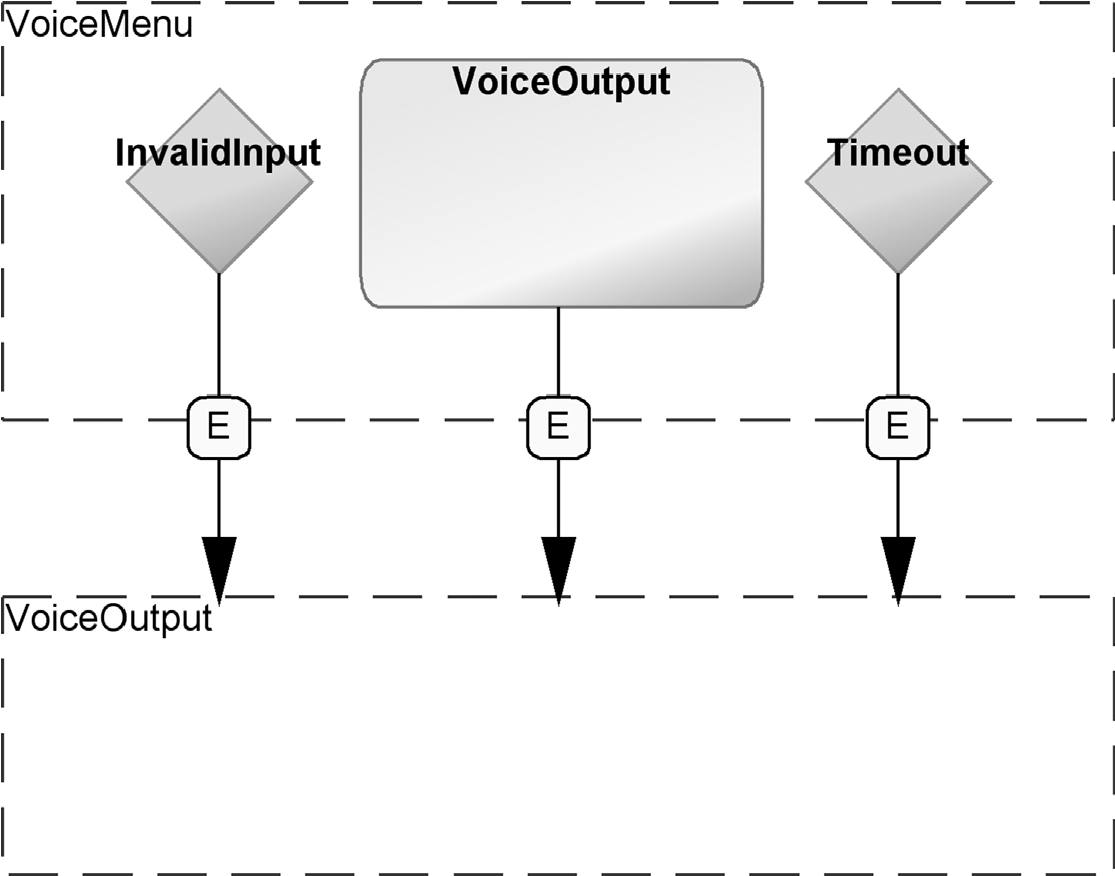


FIGURE 7.2 VoiceMenu elements with a VoiceOutput subdiagram

that is, there will be one “Invalid input” subdiagram, and each InvalidInput relationship will explode to that same diagram. The same applies to Timeouts, but each VoiceOutput object will generally have its own VoiceOutput subdiagram. As there is a limit to the complexity of a usable voice menu, no need was envisaged at this stage for an element in a VoiceOutput subdiagram exploding again to its own VoiceOutput sub-subdiagram.

The concepts of the lower-level VoiceOutput modeling language are shown in Fig. 7.3. The main elements of the language are the Text and SystemCall object types. A Text represents a sequence of TextFragments played one after the other, with no variation. A SystemCall represents a sequence of system commands: register assignments, special speech commands, and so on. The TextFragment and Command objects can only be used inside Text and SystemCall objects, not directly in the model itself.

In order to specify more complex flow control than simple sequential chains of speech and system commands, the language provides conditionals jumps with If and GotoPoint objects. An If object is made up of an Test such as “A >=” and a Parameter containing the value to be compared with. The condition can be inverted through the Boolean property, Not.

The elements in the diagram are mostly connected into a sequential chain of Flow relationships. As the top row of Fig. 7.3 shows, such a Flow can come From several different kinds of objects and go To a slightly different set. Start and Stop can only be in appropriate roles in such a flow, and If cannot be the source of a normal Flow relationship. Instead, If has two different relationships leaving it: True and False. If the result of the whole condition is true, control flow will jump to

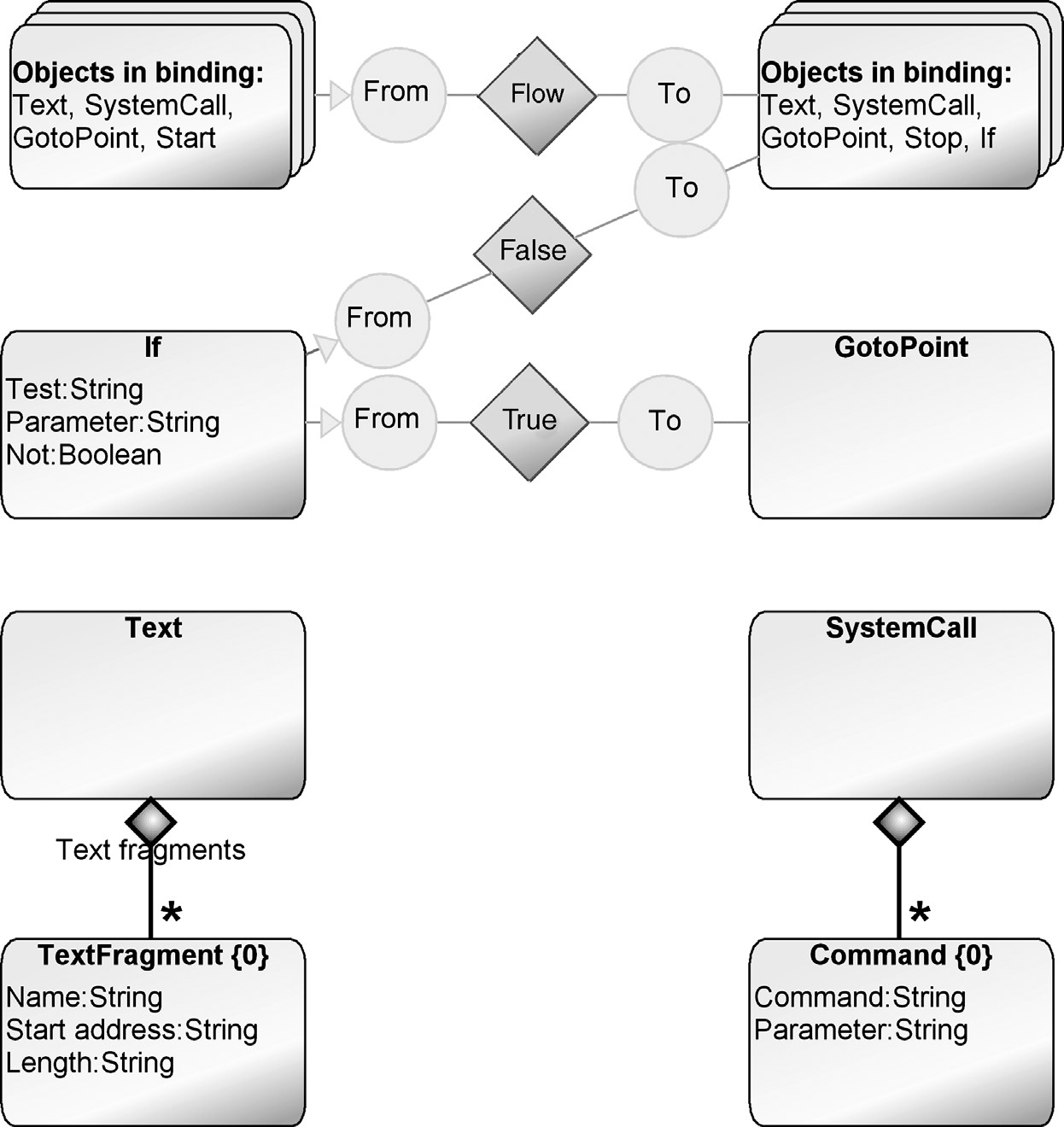


FIGURE 7.3 Lower-level metamodel, VoiceOutput

the GotoPoint at the other end of the True relationship. If the result is false, control flow will follow the False relationship to any of the normal target objects, just as in a normal Flow.

The If construct is thus not a full if..then..else familiar from third-generation languages, but a simpler conditional jump, as is common in assembly languages. GotoPoint has no behavior of its own: when included in a normal flow sequence control simply passes on to the next element. Instead, it serves simply as a label, the target for an If jump.

Once again, there are the normal rules for Start: one instance per graph, and only one From role per instance. This time, a similar constraint on From and To roles applies to most of the object types: only GotoPoint and Stop have no such restrictions. We can allow multiple Stops and multiple incoming To roles for each; the metamodel already prevents From roles leaving it. An If object should have only one True and one False relationship leaving it.

7.3.2 Possible Improvements

As this modeling language was made in such a short time, and has not been developed further, it is worth looking at some areas in which it could be improved. Some of the names for concepts could be fine-tuned, for example, GotoPoint might be better as “Jump Target” or “Label,” and Text should perhaps be “Speech.” These are however minor points, and easy to address at any stage—although it is worth noting that with some tools, changing the names of concepts can have catastrophic consequences: the next time the model is loaded, all instances of those concepts may disappear.

Perhaps the clearest problem is the repetitiveness of the InvalidInput and Timeout relationships in the VoiceMenu models (see Fig. 7.5). If in most cases the same structure will be in a model, but there may be some variation, it is better to assume the default and only use the structure where the model should behave differently. The norm for InvalidInput and Timeout is to return to the previous VoiceOutput after saying a simple message, so that should be taken as the default if these relationships are not specified. This would make the models faster to build, smaller, and easier to read, at no expense in terms of expressive power.

Another difficulty is specifying a watertight constraint for the False relationships leaving If objects. Most objects simply have one From role leaving them, but If can have two, one for True and one for False. Constraining to two From roles does not solve the problem: they could both be either True or False. We can constrain If to have only one True relationship, but the case of False is harder.

If we say an If can be in only one False relationship, we also exclude the possibility of an If followed by an If: the second If takes part in two False relationships, one incoming and one outgoing. The simplest solution would be to make the leaving role for the True case different by creating a new role type, for example, JumpFrom. This would allow us to specify that each If can be in at most one JumpFrom role (for True), and at most one From role (for False).

Looking at the False relationship, however, there may be more that we can do. The False relationship is actually no different from a normal Flow. It would probably be better to have just a normal Flow relationship for it, and add If to the set of source objects at the top of the figure. The True case would be distinguished by a different role. Since GotoPoint can allow several incoming roles, we should probably distinguish those in the normal sequential flow (To) from those that are jumps to this label (JumpTo). That would allow us to constrain GotoPoint to a single incoming sequential role—corresponding to the single line of assembler that can precede the label—but many incoming JumpTo roles corresponding to conditional jumps to the same label from multiple places in the code.

This would make the rest of the constraints more similar and allow the use of inheritance. Rather than having to specify constraints for each object type separately, we could give Text, SystemCall, If, and GotoPoint a common Abstract supertype, and

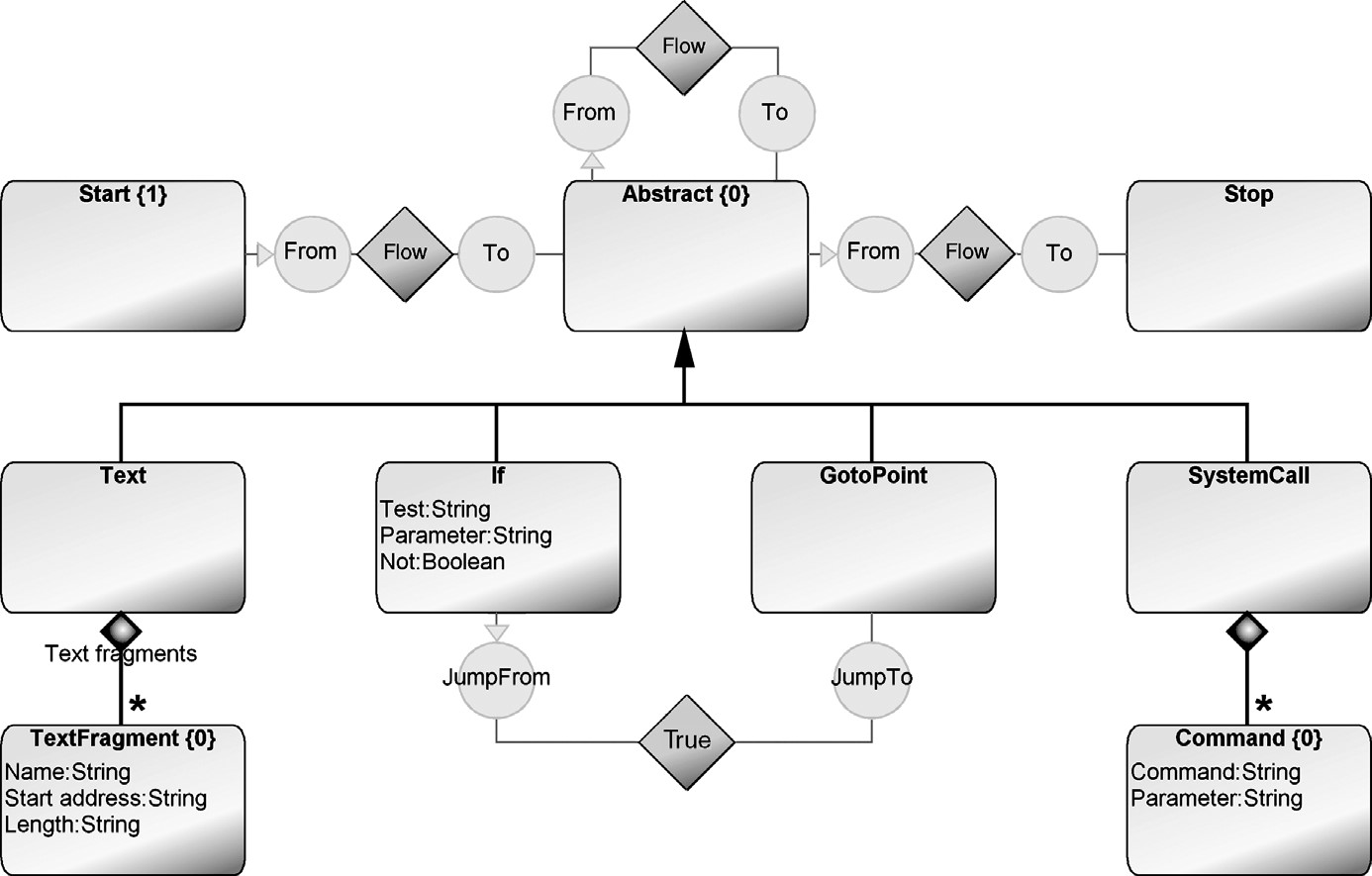


FIGURE 7.4 Alternative metamodel for VoiceOutput

specify the supertype in the bindings and constraints. This way there would only be three constraints: any Object could be in at most one From role, Abstract could be in at most one To role, and If could be in an most one JumpFrom role. The result of these changes would look like Fig. 7.4, but we will stay with the original metamodel for the purposes of this chapter.

7.3.3 Modeling Notation

As Domatic already used some simple flow chart symbols, the notation took these as its basis. Start and Stop were gray boxes containing their name, and conditional points such as DTMF\_Input and If were represented as diamonds. In the toplevel VoiceMenu diagrams, the schematic symbol for a loudspeaker was used to represent all items containing speech—VoiceOutput, InvalidInput, and Timeout. This was partly a concession to time limitations: separating these symbols only by color might be confusing, particularly since one is an object and two are relationships.

In VoiceOutput diagrams, speech segments were represented by a cartoon speech bubble showing the sequence of words or phrases to be spoken. SystemCall sequences were shown with a traditional flowchart symbol: a cut-off rectangle. The lines from If were labeled as TRUE and FALSE, with TRUE leading to a circle representing the GotoPoint.

7.4 HOME AUTOMATION MODELING LANGUAGE IN USE

For each home automation system type, there would normally be one set of diagrams specifying how the user could control it over the phone. At the top level would be a VoiceMenu diagram, and each VoiceOutput object in that could be exploded to its own VoiceOutput diagram.

7.4.1 Example Models

In our example application in Fig. 7.5, the telecom module responds to the call with the main menu: an initial welcome message and list of the options. To keep things simple, here there are only two options: pressing 1 takes the user to the mode menu and pressing 2 to the version info. The version info is simple: it reads out the version info and waits for the user to press 1 to return to the main menu. The InvalidInput and Timeout relationships are even simpler: each simply says “Invalid input!” or “Timeout!” and returns to the previous menu as shown.

The mode menu is more complex: it reads out the current mode and a list of all modes, telling the user which key to press for each. The DTMF\_Input for the mode

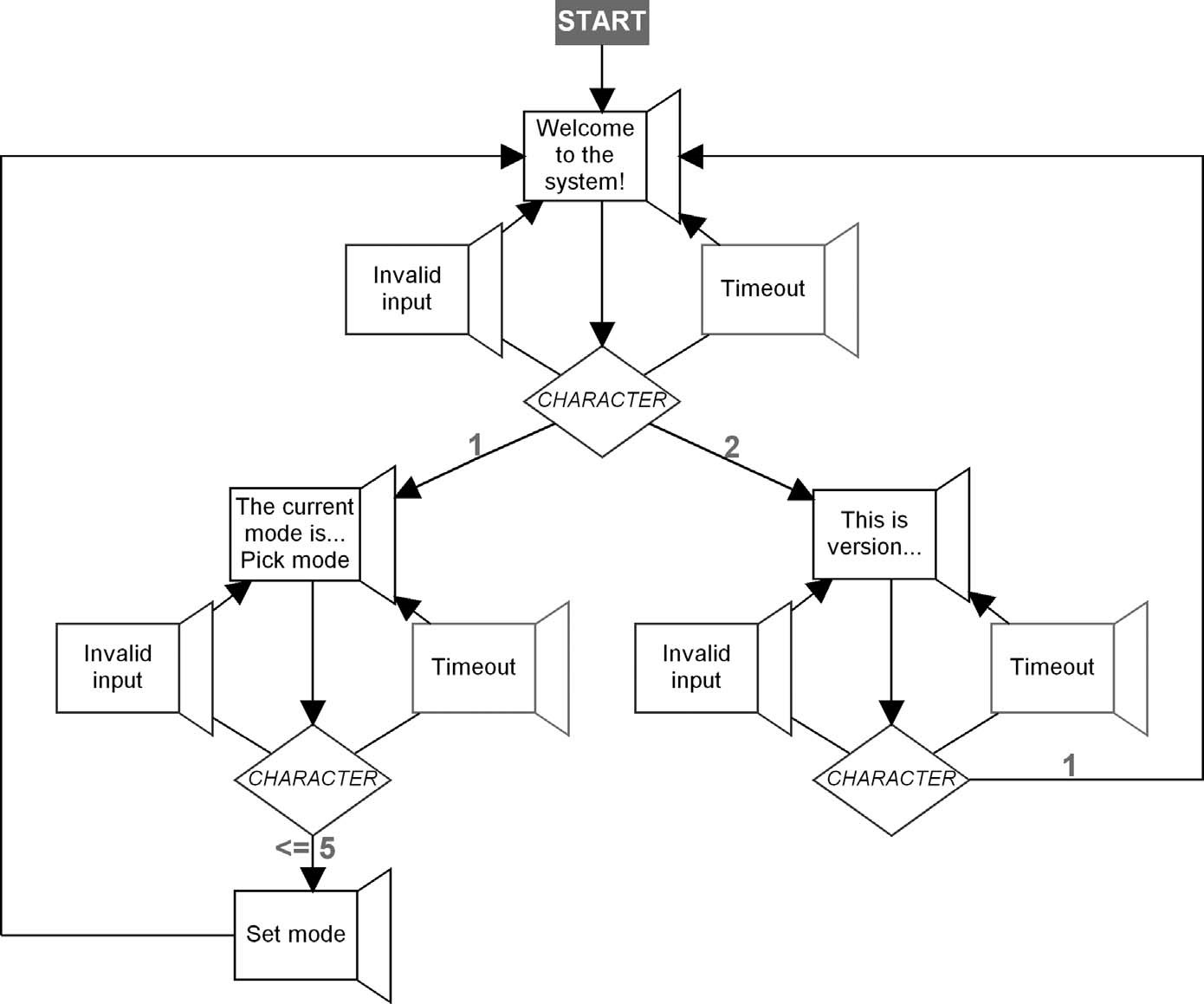


FIGURE 7.5 Sample VoiceMenu model

### HOME AUTOMATION MODELING LANGUAGE IN USE

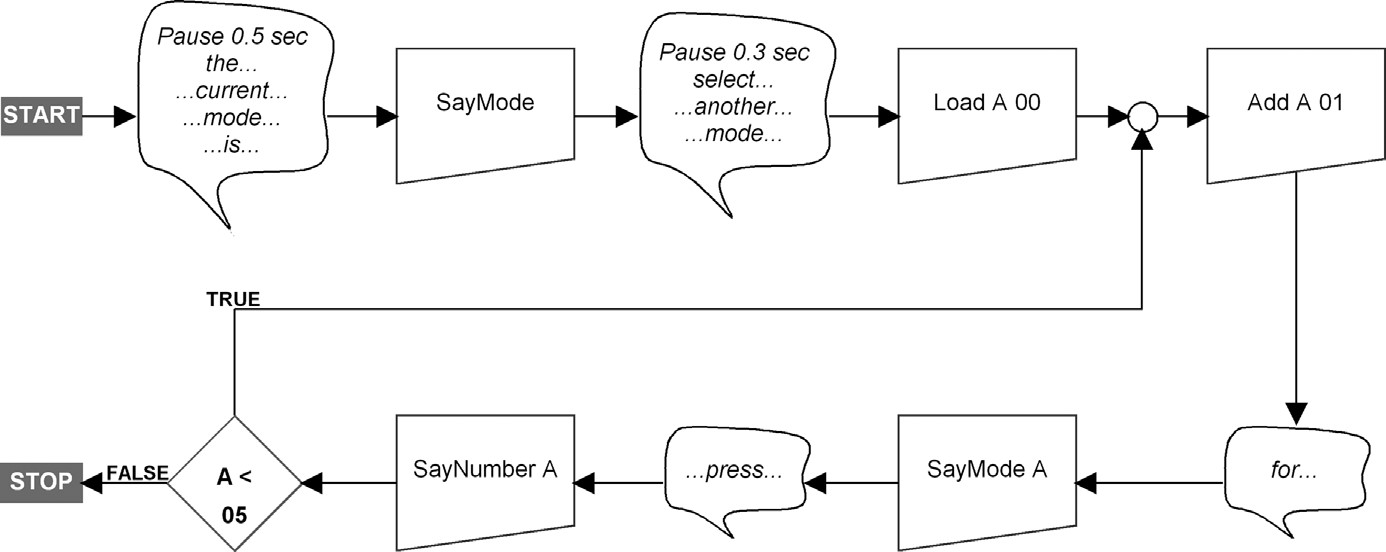


FIGURE 7.6 Sample VoiceOutput model for mode menu

menu allows the user to press a key corresponding to a mode. As there are five modes, the key must be from 1 to 5 (setting the mode to 0 does nothing). If legal input is received, the “Set mode” object’s subdiagram uses a SystemCall to change the system mode to match the key which was pressed.

The details of the mode menu are described in the VoiceOutput subdiagram in Fig. 7.6. After initially stating the current mode and telling the user to select another mode, the application initializes a counter variable, register A, to zero. After the GotoPoint, A is incremented and we move to the bottom row of the diagram, heading left. The number of modes, five, and their names are built into the system, so the system can say the name of the first mode and “press 1.” In the If object we check that A has not yet reached the value of the last mode, five, and if so we jump to repeat for the next mode from the GotoPoint. After the fifth mode has been read and the test in If fails, we exit via Stop back to the VoiceMenu diagram, where we wait for the user to press a key corresponding to a mode.

7.4.2 Use Scenarios

As we mentioned above, the two modeling languages aimed to provide a natural way to describe and specify the desired behavior of the voice menu, and of the voice elements and system calls that made up each segment of speech. The VoiceMenu language was specific to the domain of voice menus, which was a common basis shared by Domatic’s developers, clients, and clients’ end-users. It could thus easily be used as a medium of conversation between all these stakeholders, and allowed specification of systems at a high level of abstraction.

A likely use scenario would have been for a Domatic employee to work with a client to design the voice menu, directly using the VoiceMenu modeling language in the DSM tool. If example texts were specified in the top-level elements, a slight modification of the generator would allow working prototypes to be built and tested immediately.

The VoiceOutput language was based on the domain-specific side of the in-house assembly language, which was in turn based on the features offered by Domatic’s hardware platform. This modeling language was designed for use by Domatic’s developers, although simpler cases could be handled easily by nontechnical personnel. In the current state of the language, the direct inclusion of assembly language commands would have made using the whole language on more complex cases too complicated for anyone unfamiliar with the assembly language. Looking at more examples of the usage of that language would probably have allowed the use of higher-level constructs, for example, to replace the three steps in the above model— Load A 00, Add A 01, and A < 05—with a simpler single “For A = 1 TO 5” construct.

As things stood, the modeling languages would allow the creation of the complete range of applications that existed for that framework. Speech elements could be reused across multiple models, keeping memory requirements down in the finished product. As part of this reuse, it would be useful to know the total set of speech fragments used in a given application. This could be produced by a generator, guaranteeing that the set of samples for a product included all of those that were needed, and only those.

Often with reusable components, it is also useful to create an explicit library of reusable components. This helps prevent developers inadvertently reinventing the wheel because they did not know of the existence of a previously made component. This could be accomplished with a simple little modeling language that would contain a set of TextFragments. When developers wanted a text fragment, they would pick it up from the library, adding it there if nothing suitable existed. An example of such a library is shown in Table 7.1. Here we have included the start address, to fit with

TABLE 7.1 Library of Text Fragments

|  |  |  |
| --- | --- | --- |
| Label | Start address | Length |
| ...press...  Pause 0.5 sec the...  ...current...  ...mode... ...is...  Pause 0.3 sec select... ...another...  for...  Timeout!  Invalid input!  Welcome to the system!  Press 1 for mode menu  Press 2 for other menu  Welcome to the other menu  Press 1 for the main menu This is VoiceMenu v1.2 | 000 001 002 003  004  005 006 007  008  009 010  011  012 013  014 015  016  017 | 1  5  1  2  1  1  3  2  3  1  3  5  8  7  8 10  11  11 |

### GENERATOR

Domatic’s practice. In actual use, it would probably be better to omit this and instead have the generator automatically create sequential numbering separately for each product. A useful addition would be a property for each fragment that pointed to the actual sound file, so developers could listen to it directly from the model and even emulate whole sections of speech.

7.5 GENERATOR

The generator produced the necessary code for the whole application in the assembly language that Domatic used. As the generator was based on existing best-practice

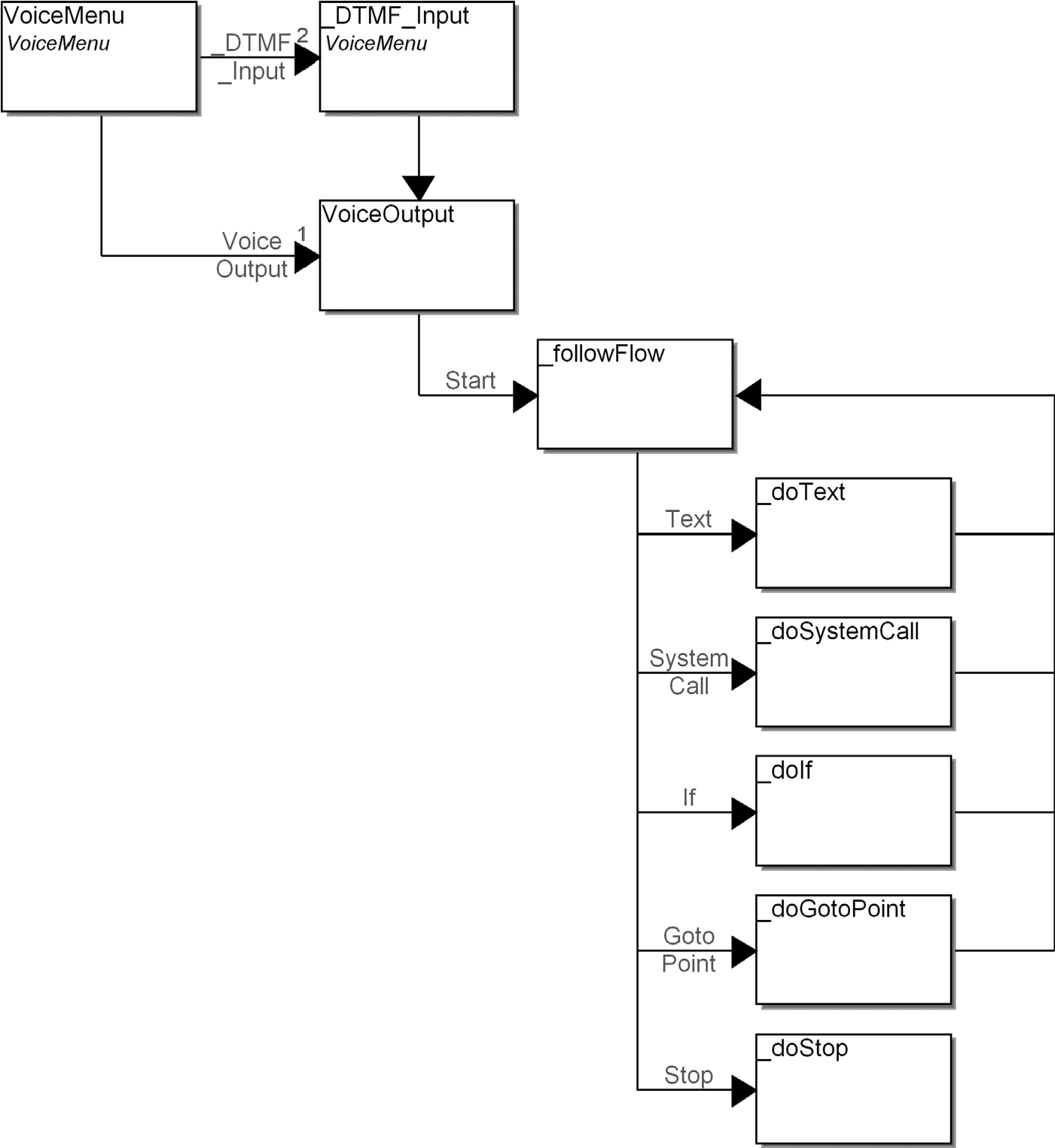


FIGURE 7.7 Home automation generator structure

code, the output was virtually indistinguishable from handwritten applications. One concession was made to the time constraints: it would have been hard to generate the correct absolute memory addresses for jumps, as this would have required calculating the byte length of each assembly instruction. Instead, labels were generated as part of the output, and jumps were directed to the labels. A quick change to the assembler made these jumps function properly.

7.5.1 Generator Structure

The generator was divided between the two modeling languages in the obvious way. Figure 7.7 shows the parts of the generator and the calls between them.

At the top level, a VoiceMenu generator started off the generation for the top-level VoiceMenu diagram, iterating over each VoiceOutput object and each \_DTMF\_Input object. The handling of DTMF input, invalid input and timeouts was all generated at this level. For the sample VoiceMenu from Fig. 7.5, the code output for the first VoiceOutput and DTMF\_Input is shown in Listing 7.2.

Listing 7.2 Generator output for first VoiceOutput in sample VoiceMenu.

:3\_266 */\* code for "Welcome to the system!" \*/*

Say 0x12 8 'Welcome to the system!'

Say 0x13 7 'Press 1 for mode menu' Say 0x14 8 'Press 2 for version info'

:3\_306

GetDTMF Timeout 5

IfNot

Jump 3\_354

Test DTMF = 1

If

Jump 3\_450

Test DTMF = 2

If

Jump 3\_468 Say 0x11 5 'Invalid input!'

Jump 3\_266

:3\_354 Say 0x10 3 'Timeout!'

Jump 3\_266

:3\_450 */\* code for "The current mode is... Pick mode" \*/*  ...

:3\_468 */\* code for "This is version..." \*/*  ...

The first block, labeled 3\_266, is specified in the lower-level VoiceOutput subdiagram. 3\_306 is the start of the DTMF handling: wait for DTMF input, timing out after 5 seconds (this period is taken from the Timeout relationship). If

### GENERATOR

we time out with no input, jump to 3\_354, say the voice output specified in the Timeout relationship’s reused VoiceOutput subdiagram, and jump back to the start of the first block.

The meat of the DTMF handling is a series of Test-If-Jump blocks, which compare the DTMF tones received with the characters specified in the ConditionalFlow relationships, jumping to the appropriate VoiceOutput block if there is a match. If no match is found, we fall through to the “Invalid input!” section at the end of the 3\_306 block, which is generated similarly to the Timeout section.

Moving on to the VoiceOutput section of the generator, Listing 7.3 shows the output for Fig. 7.6. The first block shows the code for the flow up to the GotoPoint, which is label 3\_844. The code after 3\_844 is essentially the sample code from Listing 7.1: the generator fulfils its requirements.

Listing 7.3 Generator output for Mode menu.

:3\_450

Say 0x01 5 'Pause 0.5 sec'

Say 0x02 1 'the...'

Say 0x03 2 '...current...'

Say 0x04 1 '...mode...' Say 0x05 1 '...is...'

SayMode

Say 0x06 3 'Pause 0.3 sec'

Say 0x07 2 'select...'

Say 0x08 3 '...another...' Say 0x04 1 '...mode...'

Load A 00

:3\_844

Add A 01 Say 0x09 1 'for...'

SayMode A Say 0x00 1 '...press...'

SayNumber A

Test A < 05

If

Jump 3\_844

The VoiceOutput generation was handled by a generator in the VoiceOutput modeling language. As the basic sequential flow control was the same for all object types, that was handled with one generic \_followFlow generator. This followed the relationship to the next object and called the generator for that object type. The name of the subgenerator to be called is formed on the fly from the name of the type, allowing a new type to be added to the modeling language simply by specifying one small subgenerator for it. Listing 7.4 shows the generator definition. A From role, Flow or False relationship, and To role are followed to the next object: (A|B) specifies either type A or type B, and () specifies any type. The name of the generator to call is formed from the output between the subreport and run keywords, that is, \_do followed by the name of the object’s type, for example, \_doIf.

Listing 7.4 “\_followFrom” generator definition.

Report '\_followFlow'

do ~From>(Flow|False)~To.() { subreport '\_do' type run } endreport

The generator begins from the Start object and calls \_followFlow from it. Each object type’s generator handles its own line or lines of output and then calls \_followFlow again, making a simple recursion step through the model. The generator for the Stop object does not recurse, and thus the generation finishes there.

The \_doIf generator handles the FALSE path as normal sequential flow with the \_followFlow generator. The TRUE path, however, is handled by outputting an assembly language test statement followed by a conditional jump to the label specified by the GotoPoint, without any recursion to follow the code on from the label. This avoids the problem of an infinite cycle or duplicate code (see Section 11.3.5 for more on this issue in general).

Text and SystemCall objects may contain several TextFragment or Command objects, leading to several lines of output, for example, the first five lines of Listing 7.3. These are handled simply by iterating over the contained objects as shown in Listing 7.5. The listing also shows the final \_followFlow call.

Listing 7.5 “\_doText” generator definition for Text object type.

Report '\_doText'

do :Text fragments {

' Say ' :Start address; ' ' :Length ' '''

' Say ' :Start address; ' ' :Length ' '''

:Label '''' newline }

subreport '\_followFlow' run

endreport

### MAIN RESULTS

7.6 MAIN RESULTS

The presentation by Domatic’s experts to their management revealed that the proof of concept was partially successful. The modeling language and generators were seen to have accomplished the objectives set for DSM:

. The models visualized application structures well

. The modeling language and generators forced developers to do things right

. Applications which previously took a day could be made in an hour or two

. Better reuse possibilities within and across products

. The models provided consistent documentation

. Test plans could be integrated with models

However, Domatic’s needs were not just for this single domain, but for applying DSM in a range of domains. The time constraint of effectively three hours meant that only simple cases had been handled in this domain, leaving uncertainty over whether DSM or the tool could cope in other domains. It is also likely that time constraints forced the consultant more into doing things himself, rather than having the time to explain everything and let Domatic’s experts try their hand at doing things themselves. Developers are loath to tell management something can be done unless they are certain of their own ability to do it. Domatic’s experts thus also raised a number of concerns that the proof of concept had not been able to answer:

. Coding such simple applications was not challenging for average developers

. Uncertainty about 100% code generation

. Uncertainty about backward compatibility with existing code . Are code generator facilities flexible enough?

These are indeed often concerns after a proof of concept: even in the best case, in two days it is simply not possible to prove these beyond doubt. Normally, however, sufficient progress has been made that the experts feel these have been demonstrated as well as could be in the limited domain, and certainly beyond what they themselves had expected.

The extra two hours of work after the workshop are obviously not reflected in this bullet list from the presentation. This is unfortunate, as those two hours addressed three of the main concerns. First, the refactoring of the modeling language allowed the creation of more complex applications. Second, finishing off the code generator proved its ability to generate 100% code. Finally, these extra parts of the generator required some of the more powerful features of the generator facilities, demonstrating flexibility that had not been needed in the earlier parts.

7.7 SUMMARY

This case provides a good example of how DSM can be used at the lowest end of the abstraction spectrum, generating assembly language for an 8-bit microprocessor directly from high-level models. It is interesting to see that even here, the productivity increase fits within the normal range of a factor of 5–10. At first sight, there seems little benefit in replacing a line of code ‘Say “Press”’ with an object containing the word “Press.” Looking more closely, we notice that in both cases we need to also specify the starting address and length of the speech sample. If we were writing the assembly language by hand, we would have to look these up somewhere and copy them into the code. Here we can simply reuse an existing “Press” TextFragment object, and should any of its details change later, those changes will be automatically reflected here. Similarly, rather than having to fathom our way through the spaghetti of Goto jumps and labels, the models make the control flow instantly and intuitively clear.

The fact that the modeling language and generators were built in such a short space of time also provides good evidence for the use of efficient DSM tools. It is particularly important to be able to quickly iterate through several cycles of adding a small part to the modeling language and testing it, without existing models being rendered unusable.

We can learn several lessons from the less successful parts of the case. With regards to the first, abortive, day of the workshop, there is always a danger when trying to build a modeling language that we invent something that is really more of a model, or simply allows us to draw architectural diagrams. A good rule of thumb to apply is whether we can realistically generate code from the models made with this language (assuming code generation is desired). Another test is how many models of this kind would we be likely to build. If the answer is “one diagram per product”, the language does not go deep enough.

Second, even if people talk about the choice of example domain not being important, it still has a large effect on how the overall feasibility of DSM is perceived. Seeing DSM work in one domain holds little credibility, if that is not a domain the customer intends to use it for. People are naturally afraid that the actual domain they want will turn out to be more complicated than the example shown. Looking at examples from other companies helps little: for instance, Domatic were familiar with Nokia’s use of DSM, but dismissed cell phone development as not being true embedded development—far too simple. The truth of the matter is that most development, when looked at from 10,000 feet, is simple: the complexity is often in the languages, frameworks, and tools we use. The power of DSM is that it can leverage the abilities and experience of an expert developer to create a development environment that more closely resembles the view from 10,000 feet.

Finally, DSM is a hard sell to companies who do not control their own products. If the problem domain changes with each client, even proving that DSM works for an existing range of products will rarely be enough. Instead, before the company starts using DSM, it must be convinced that it can itself build a new DSM language and

### SUMMARY

generators for a sizable proportion of the new domains it may find itself in. This will in many cases be perfectly possible, but the belief in its possibility can often only be acquired by experience from a few successful applications of DSM. This cycle of chicken and egg will make DSM a nonstarter for this kind of company, unless they possess an unusually high breadth of experience or depth of understanding to see and understand how DSM works in both theory and practice.