### CHAPTER 11

## GENERATOR DEFINITION

Building a generator is one of the two main tasks when creating a Domain-Specific Modeling (DSM) solution. The idea of code generation is by no means new: most developers will be familiar with the fixed generators supplied with generic modeling tools, and many readers may well have written their own little scripts or macros to generate repetitive code. A DSM generator is closer to the latter, because it is built by you for your specific purpose, rather than by a tool vendor aiming for the widest possible market. It must however go further than most scripts or macros, since it is intended for automated use by people other than its creator, and its output will not normally be edited—or even checked—by hand.

Although a DSM generator must aspire to higher quality than ad hoc scripts and macros, a problem in a DSM generator is less severe than with fixed generators supplied with modeling tools. Unlike these prepackaged generators, you remain in control, and so can fix the problem—on your timetable rather than when it happens to suit the vendor. DSM generators also solve the problems of the “sausage factory” generators, which churn out multiple similar blocks of boilerplate code to be hand edited. Unlike those, a DSM generator can later be changed and all code regenerated: the equivalent of changing your recipe and having all sausages already delivered to shops update automatically!

For most developers, one of the hardest tasks in Domain-Specific Modeling is to not build the generator. Why would we want to avoid building the generator: surely that is a vital part of a DSM solution? Absolutely—and it is also one dear to the heart of developers, being close to the code they are used to writing. Thanks to the

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

267

overenthusiastic hyping of 1980s and 1990s CASE tools with their fixed modeling languages and “one size fits all” generators, the generator is also the area that many developers are most skeptical about.

These two factors lead many to want to jump in and build a generator right near the start, often hand in hand with developing the modeling language. Unfortunately, such solutions almost invariably turn into languages that describe code rather than things in the problem domain. The result looks like a bad rehash of UML, without even the chance to claim it is a standard. The level of abstraction is not raised, and there is no real way to generate full code from such models.

As we build our modeling language to take us to a higher level of abstraction, there can thus be no premature slipping back down the slope into code. When we have the language firmly established on a higher level of abstraction, it is refreshingly easy to build a path back down to the code level. The information from the models seems to flow down naturally, as if by some gravity of abstraction, into the form needed for the compiler. This ease should come as no surprise: it has always been easier to build a compiler than a reverse compiler.

Developers are often afraid that they will produce a modeling language that would have no sensible, easily performed mapping to code. While in theory this might be possible, in practice any expert who has coded applications in the domain would be most unlikely to do this. They will find themselves making decisions in the modeling language that, in spite of their best efforts, are at least informed by their experience of patterns of code in that domain. While that tendency and experience are of course valuable, it is still worth fighting them down to a reasonable extent. If the modeling language can be made 10% better to use, at the expense of making the creation of the code generator 10% harder, many modelers will benefit over many applications, while only one metamodeler needs to suffer once.

Still, as metamodelers we are all for reducing the suffering of fellow metamodelers where possible! One important way to do that when building generators is to have something concrete to work with: a working mini-application in exactly the format you want to generate. That is what we will look at in the next section, before moving on to the generators themselves.

11.1 “HERE’S ONE I MADE EARLIER”

Before we can build a generator, we need to know what we want to generate. The best way to do that is to have a working example of the output. Remember, DSM is about automating what we would otherwise be doing by hand. Unless we know what we want and are able to do that by hand, there is little hope of teaching somebody else to do it— let alone teaching something as stupid as a computer to do it. The good news is that we only need one example of each part of the output, and the computer learns quickly.

But wait! It is not enough to have just the output: since a generator is a kind of transformation, we must also have the corresponding input. We can either build a model that roughly corresponds to some existing code, or then build a small representative model and corresponding code.

### “HERE’S ONE I MADE EARLIER”

Let’s assume that you have no code to start with, so first you want to build a model. Such a model might have one instance of each of the three or four main object types. Sketch out an implementation of that which would work with your current code, and check that the model can give you the information you need. If it is at least close, that will be fine. You can allow yourself some assumptions for now: maybe an object type could map to one of two different kinds of code, but just choose the more common solution for now. Go ahead and write the application and get it to run. Aim to write code similar to current best practice in your applications, but do not waste time striving for perfection or minimalism. The best code at this point is the kind that you could explain easily to a new employee, not the kind that will save two bytes or ten milliseconds because of a clever trick with a certain version of a library. But have no fear: if you can specify when to perform that trick, you can later add a condition to that part of the generator and reap the saving every time it is possible.

Now you have a matching pair of a model and the desired output. 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 fixed 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 and the ease of use of the modeling 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. When outputting a name from the model, we may thus need to apply a filter to it that replaces nonalphanumerics with underscores.

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.

This section has described the basic process of building a generator, regardless of the kind of model, output language, and language for writing generators. To give more detailed advice on building generators, we first need to know what kind of generator language we have.

11.2 TYPES OF GENERATOR FACILITIES

Czarnecki and Helsen (2003) identified two main approaches for generating code or other text from models: visitor-based and template-based. In practice, there are at least a couple more ways, ranging from simple programmatic access to the models to crawler-based generators outputting multiple streams, and even generators that are themselves the output of other generators.

11.2.1 Programming Language Accessing Models Through an API

The minimal facility necessary for generation is programmatic access to the models. In an open source modeling tool or model repository, the programmer can code generators directly against the model data structures. While initially such direct access may feel welcome, the low-level programming it requires, together with the unpleasantly high coupling between the generator and modeling tool, mean this is normally only used where an organization has hand coded the modeling tool themselves. Even then, as the tool matures the benefits of separating the implementation of modeling and generation become apparent. The need for higher-level commands to read, navigate, and output model structures also quickly leads to a separate generator component.

Direct access to model data structures also forces the generator to be built in the same programming language as the modeling tool and to run in the same memory space. To solve these problems and alleviate those mentioned above, tools often offer an Application Programming Interface (API) for model access. Most early APIs limited the generator to running on the same platform, and indeed the same machine, as the modeling tool, but more recent work removes these restrictions.

An external generator API can be either message-based or data-based. In the former, the model data remains in the modeling tool, and the tool’s API only passes pointers or proxies for those objects to the generator. The generator can still be written as if it were operating on local data, but all calls are actually proxied through to the modeling tool, and only further proxies or primitive data types like strings or integers are returned.

In data-based APIs, calling an API function returns the actual data structures, or rather a copy ofthem,tothegenerator.This requires theduplication ofthe data structure typesand functions in thegenerator. Italso bringsthe problem of where tocutthemodel up: if the generator requests a graph, do we return the whole data structure of the graph, all its objects, all their subgraphs, and so on? If we do not, there must be some method to determine which parts of the data are to be returned in full and which as proxies.

A well-made API of either type will behave similarly to the other type: no long delays and running out of memory in a data-based API, or apparent instant access to all data in a message-based API. Both types are, however, only approximations. What is probably worse, though, is that both only offer standard generic programming languages to read, navigate, and output models.

While there are good programming languages for manipulating complex networks of objects, and good programming languages for transforming one text stream into another text stream, there are few if any generic programming languages well-suited to navigating a complex network of objects and outputting text. Hence the need for the languages and facilities specifically designed for generation, which will be covered in the rest of this section.

11.2.2 Model Visitors and Model-to-Model Transformations

The simplest kind of generator facility is a model visitor, which makes a mapping from structures in the modeling language to structures in the output language. Normally, each type in the modeling language is mapped to an output language structure. For example, a graph type “Watch Application” may be mapped to a class, an object type “State” may be mapped to a method, and another object type “Time Variable” may be mapped to a field.

In some cases, the mapping is focused on the modeling language, with a fair amount of freedom in what each type can map to. The generator visits each element in the model, calling the generator for that element’s type via the Visitor pattern.

In other cases, such as XMF-Mosaic, the mapping is made in three stages. The first stage maps the model elements to an intermediate set of concepts corresponding to a broad kind of programming language—most likely object-oriented languages. The second stage maps each of these generic object-oriented concepts to the corresponding concept in a particular object-oriented language. Finally, the third stage generates the relevant code for that concept in that language. Such an approach would be useful if a company needed to generate the same application in a growing number of object-oriented languages: they could add new second- and third-level mappings as the need for those languages appeared. However, it is limiting if the need is only for one or two languages: the generator must follow the patterns and structures laid down by the tool vendor.

The set of output language structures supported for mappings would normally include the major higher-level structures such as functions, classes, and modules. If strongly linked with a particular Integrated Development Environment (IDE) or IDEs it could also extend up to projects. If linked to a particular kind of language, it could also extend down to control structures. Building a mapping for such extended model visitors would however become increasingly difficult, as the various parts of, say, a C for loop must each be mapped to some structure in the model.

A model visitor can cope reasonably well for simple skeleton code generation from modeling languages like UML into the object-oriented programming languages it represents. For DSM languages, single-stage model visitors normally fall short, unless the output itself is a DSL with a similar semantic structuring to the models. The basic idea of a model visitor is however useful in all types of generation facilities.

Model-to-Model Transformations Rather than aiming to produce textual output, a generator can also create or alter a model. This still leaves us with a model, which will need a further generator to produce the textual output required by compilers and other existing implementation tools. It is generally a particularly poor idea to create model transformations that produce models that users are still expected to edit, for the same reasons as generated code should not be edited. Some of the issues with editing generated models in the context of MDA were covered in Section 3.3.3.

The value of model-to-model transformations is thus better realized when they form one part of a chain of transformations, resulting eventually in textual output, and whose intermediate stages are invisible to the modeler. The decision of whether to generate in one step from models to text, or in several steps with intermediate transient models, is thus one that can be made by the metamodeler based on the tools available. If for instance the generation tools are not powerful enough to support a mapping from models to the required text in one step, intermediate phases may be useful. Similarly, if there already exist tried and tested transformations from a certain model format to code, it may be useful to translate DSM models to that model format. Particularly where these transformations cross tool boundaries, however, there is a danger of information being lost or twisted along the way, like a game of Chinese Whispers.

11.2.3 Output Template

An output template consists of the code you want as an output, but with parts that vary based on the model replaced by commands surrounded by some escape character sequence. This approachis familiar from web pages built dynamically on the client side with HTML containing JavaScript commands, or on the server side with PHP or ASP.

Template-based generators are probably the most common kind, and one of the oldest. Examples include JET (Java Emitter Templates), the Microsoft DSLTools T4 engine, and the CodeWorker scripting language.

JET operates on input data consisting of a single Java object, and its command sequences are written in normal Java. It can only be run as part of Eclipse and is only designed for outputting Java classes. It uses <% and %> to escape its command sequences—the escape characters can be configured—and also <%¼ and %> to delimit a Java expression whose result is appended directly to the output. JET is run in two phases: translation and generation. The first phase translates the template to a Java program that will perform the actions specified in the template. The second phase invokes this program on the input to generate the output. The name of the file to be generated is specified in the first JET tag.

T4 operates on input data consisting of a single C# object, normally a DSL Tools model, and its commands are written in C#. It uses <# and #> to escape its command sequences, <#¼ and #> for direct output, and <#þ and #> to escape whole functions or other class features.

The CodeWorker scripting language operates on input data consisting of a tree data structure, formed from the parse tree produced by a CodeWorker BNF template from textual input. It uses <% or @ to delimit its commands, which are in CodeWorker’s own textual DSL. As CodeWorker uses its own language, it also has to provide its own library of basic functions for manipulating strings, numbers, files, and so on.

One problem with an output template is that it only allows output to one file or stream. The structure and order of the generation is thus forced to follow that of each output file, with one generator per output file. This is inefficient in cases where the information needed for more than one file can be found in the same place in the models. The simplest case of this is .h and .c(pp) files in C(++) generation, but other examples abound even where the output language is not so quaintly repetitive.

For instance, for each source code file we output, we often want to add a line to a makefile. It is most convenient to be able to append that line to the make file at the point where we start to generate the corresponding source code file. Using output templates, we would be forced to duplicate the navigation code that brought us to the appropriate points in the model.

11.2.4 Crawler: Model Navigation and Output Streams

A DSM generator has two tasks: to navigate and read its way around the model, and to output text based on the model. Model visitors require the modeling language elements to match the output format elements. Output templates force the navigation to be sub-ordinate to the output format. A third kind of generator, a crawler, gives more freedom, while still allowing these two simpler kinds of generation where they are possible.

Where an output template consists primarily of fixed text, interspersed with escaped command sequences, a crawler consists primarily of command sequences, with fixed text escaped simply by placing it in string quotes. At its simplest, then, a crawler looks like an output template, but with the escaping turned inside out.

The generator language in a crawler will normally be a textual Domain-Specific Language, as is the case in say MERL, the MetaEdit+ Reporting Language. On the borders between true generator languages and general purpose languages used for generation are languages like DOME’s Alter, which is an adaptation of Scheme. The benefits of a true DSL are similar to those of DSM itself: conciseness, a higher level of abstraction, and fewer errors.

A crawler maintains and operates on two stacks as it generates: one for model element navigation history and another for output streams. At any time, then, there is a current model element and a current output stream. This makes for a concise language: rather than have “fileWriter.write(this.name())” we simply have “name.” The details of which element’s name we want and which file we want to write it to are easily inferred from the stacks.

Similarly, crawlers can free us from error-prone explicit pushing and popping, or assigning iterator variables, which would otherwise take up a large part of the generator text. Each navigation operation works like a loop, with an implicit push at the start and pop at the end.

For instance, if we ask the crawler to navigate from the current State to each of the States directly reachable from there, the top element on the stack will initially be the current state. As we process each reachable State, it will be pushed on the stack, then the output for it will be performed. Afterwards it will be popped off the stack, and we will move on to repeat the loop for the next reachable State.

Similarly, if we are generating a number of C files and a makefile for them, we might start with the makefile on the top of the output stack. We could then navigate to each model element from which a C filewill be generated. For each element, wewould output a line to the makefile for that element’s C file, then push a new output stream directed to that C file. With that on the top of the stack, we would perform the output for the contents of the C file, then pop it off the stack before moving on to the next model element.

Using a DSL for the commands can give a major advantage over a standard programming language. This is especially so in the area of navigation, which takes up the bulk of the generator. Listing 11.1 is an example of a standard programming language, C#, used in a T4 template from the Microsoft DSLTools forum (apparently an earlier version, since it uses <% rather than <#).

Listing 11.1 Generator written in a generic programming language.

<h2>ConceptBs</h2>

<% foreach ( ConceptB b in this.ConceptA.Bs )

{

%>

<h3>Name: <%=b.Name%></h3>

<h3>Outgoing links:</h3>

<%

MetaRoleInfo roleInfo = b.Store.MetaDataDirectory. FindMetaRole (BReferencesB.ReferringBsMetaRoleGuid); if (roleInfo != null)

{ foreach (object link in b.GetElementLinks(roleInfo)) {

BReferencesB relationship = link as BReferencesB; if (relationship != null)

{

%>

<p>Refers to: <%#relationship.ReferencedB.Name%></p>

<%

}

}

}

}

%>

And here in Listing 11.2 is the same thing using a DSL, in the crawler-based MetaEdit+ Reporting Language.

Listing 11.2 Generator written in a language made for generating.

'<h2>ConceptBs</h2>' foreach .ConceptB

{

'<h3>Name:' :Name '</h3>'

'<h3>Outgoing links:</h3>' do ~From~To.ConceptB

{

'<p>Refers to:' :Name '</p>'

}

}

The key difference here is in the line in the middle, “do FromTo.ConceptB.” Starting from the outer foreach loop’s current ConceptB, it says to crawl along any From role and its To role into the next ConceptB. That one line replaces twelve lines from the version written in C# (everything between “Outgoing links” and “Refers to”).

This kind of pattern is very common in any code generation or reporting on a model. Those using standard programming languages as generators will thus quickly find their generators full of blocks of code similar to the above. Since DSM is intended to save developers from precisely this kind of unproductive code duplication, this is more than slightly ironic: the old adage of the shoemaker’s children comes inescapably to mind.

11.2.5 Generator Generators

DSM people tend to be happy with the idea of doing something on a meta level; some even get excited about it. For such people, the idea of a generator generator is positively intoxicating. Rather than drudge to build a generator that turns models into code, why not write a smarter program that would build the generator for us? This can be particularly attractive if the generator language is somewhat lacking in facilities for handling repeated similar cases: subgenerators, parameterization, reflection, refactoring tools, and so on.

The overhead in building a generator generator is however substantial. First there is the additional cognitive load of an extra meta level, and of an extra layer in which bugs can hide. Second comes the difficulty of three levels of embedded syntax: simple strings to be generated are first quoted for the actual code generator, and then the generator script containing those is quoted for inclusion in the generator generator. This situation is made worse when the three languages—those of the generator generator, the generator, and the output—all use similar punctuation and keywords. In the worst case, they might even be exactly the same language: at that point, nobody can read or write without constantly losing track of which level a particular keyword or punctuation character is on.

Finally, generator generators introduce new issues of version control. Unless the generated generators are created anew for each run, they should be versioned and mechanisms put in place to make sure all users use the correct version, and all generated code mentions the version used. If the generated generators are processed by a different facility from the normal facility—an external compiler or interpreter, say—the version of that tool must also be recorded.

These drawbacks are unlikely to deter the kind of person who is attracted by meta solutions, compilers that compile themselves, modeling languages that can be used to model themselves, and tools that were built in themselves. While the particular gene or brain chemical responsible for this attraction is yet to be identified, two things are certain. First, the effects of it are very strong and second, there is at least some correlation between its level and how smart a person is. Maybe the smartest ones are those who are pragmatic enough to recognize when not to try this approach, but few would claim they feel no attraction.

While the purest form of this pursuit is generators that produce generators in the same language, it has also been used to good effect in integrating external generation tools. The first generator exports the models in some easily digested format, along with commands to an existing external generator. The existing generator parses that format and applies those commands on it to generate the output. The commands can vary greatly in their complexity. At one extreme, they can be a simple parameterization of a mostly fixed generator, in which case the model output format must be one which that generator can read. At the other extreme, they will be a full program to run on the model output format, in which case the external generator will be a compiler or interpreter that runs that program.

It is also possible to combine both models and commands into the same file. Here the model is represented directly as data structure initializers in the language of the external generator. At the opposite end of the scale, the situation can also result in models and commands being in similar formats, for instance if the models are exported as XML and the commands in the form of an XSLT transformation, itself an XML file.

An interesting variation on generator generators occurs where the second stage of generation is deferred until the runtime of the end system. As the system is started up, it generates some parts of itself. Clearly, this requires the use of a language that is sufficiently dynamic and has good support for reflection. It also brings us neatly to our next topic: looking at various patterns of code that are often found in generated solutions.

11.3 GENERATOR OUTPUT PATTERNS

Generators produce a wide variety of different kinds of output: simple text, documentation, .ini files, XML, database definitions, and of course various kinds of code. For generation to be possible, the output required from a given generator over different models must exhibit patterns. If there is nothing an expert human can recognize as a pattern common to several examples of the kind of output you want, there is no way to generate it. Fortunately, even a glance at the kind of code most of us are forced to write these days shows there is plenty of scope for removing duplication.

Even after removing duplication, a trained eye can spot many cases where textbook patterns have been applied, particularly in well-written code.

In a similar way, there are patterns to use when writing generators. Some kinds of tasks seem to crop up often in DSM generators, and knowing a good basic solution can save significant time.

11.3.1 Simple Text

Simple text files such as configuration files or script files are generally easy to generate. They tend to fall into three categories:

1. Largely boilerplate
2. Configuration and settings files
3. Script files

The easiest are of course files that consist largely of boilerplate text, where the majority of the text remains the same independent of the model contents. Any kind of generator will cope well with these, and the template-based generators are at their strongest here.

Models that generate configuration and settings files tend to use a simple mapping between the structures in the file and those in the model. For instance, sections in an ini file may correspond to object types, with each setting and value corresponding to a property slot in that object. The crawler-based approach in Listing 11.3 would generate such a file, treating empty properties as meaning they should not be generated, and thus whatever reads the resulting file will use a default value for them.

Listing 11.3 Crawler-based generator for .ini files.

foreach .() /\* iterate over objects, expecting one instance per type \*/

{

'[' type ']' newline do :() /\* iterate over all properties of the object \*/ { if not id='' then type '=' id newline endif } newline

}

One thing that can be difficult in configuration files is that the output language, and the program that processes it, tend to have little intelligence. As the values in the file tend to be expressed in a machine-friendly format, this can require some translation from the human-friendly format used in the models. For instance, the models may specify numbers in decimal notation, but the configuration file may require them in hexadecimal. If this situation had occurred when generating normal source code, the values would probably have been output in decimal, even if earlier handwritten code had used hexadecimal: the compiler can read either with no difficulty. Now, however, either the generator language must include commands to perform the conversion, or then a separate postprocessing step must be applied to the generated file.

This is thus a strong area for generator languages that use a generic programming language for commands, as such languages will generally already have a library function for such a conversion. Domain-specific generator languages may not fare so well: even Codeworker, with its built-in library of over 200 functions, has no direct answer for this, although it can handle some particular cases. All is not lost, however, as the necessary transformation can always be written as a postprocessing step in a generic programming language. If there is no great number of such cases, they can also be made as in-line calls from the generator to such a program, making the generators slower to run but easier to understand.

Script files, such as Windows batch files or Unix shell scripts, are also generally easy to generate. On the positive side, the interpreters for these files are smarter than those for configuration files. Transformations can thus often be output as the instructions to perform the transformation, rather than require the generator to perform the calculation and output just the answer. In some cases, this can also make the generated output easier to understand: rather than contain some mystical number, with no clear relation to the model, the output shows the calculation from the model values. Even batch files are capable of this, as shown in Listing 11.4.

Listing 11.4 Using batch files for simple arithmetic.

C:\>set FahrTemp = 0

C:\>set /A KelvinTemp = (FahrTemp–32) \* 5 / 9 + 273

256

But there are surprises in store for the unwary, for instance if you try to bit-shift the above result using the >> operator, as in Listing 11.5.

Listing 11.5 Pushing the limits of arithmetic in batch files.

C:\>set /A KelvinTempHighByte = KelvinTemp >> 8

"Appends the result to the file called 8, so have to quote:"

C:\>set /A KelvinTempHighByte = "KelvinTemp >> 8"

1

C:\>set /A KelvinTempLowByte = "KelvinTemp % (1<<8)"

0

And if you put these in a batch file, you will notice that the result of the SET operation is no longer sent to standard out, and the modulus operator % needs to be doubled to avoid being interpreted as a batch file argument... Still, one of the joys of generation is learning all the little tricks and foibles of the language you are outputting!

11.3.2 Model Checking

The modeling language is the best place for rules that should always hold, and the DSM tool should offer most of the necessary types of checks that you can customize or parameterize. Where the DSM tool does not offer the checks you need, it is also possible to write a generator to check some property of a model and give feedback to the modeler.

In most aspects the generation of model checking reports is similar to the configuration files mentioned in the previous section. One difference is that the output is intended for display to the modeler, rather than for reading by a program, and for use at design time, rather than compile or run time. These factors favor the integrated generator facilities, which can then display the results of the checking as live links to the model elements. For instance in MetaEdit+ clicking on the name of an object in the generator output will take you to that object in the model, allowing you to correct the problem reported.

It may be possible to achieve something similar with other generator facilities if the checking report is output to HTML, and it is possible to make a hyperlink invoke the modeling tool to show a specific object. Some IDE-based DSM tools may also offer special integration of model checking reports, automatically running them when saving a model, and showing the results in a separate pane similar to that used for compiler errors. For instance, Microsoft’s DSL Tools take this approach, although admittedly such model checking scripts are programmed in C# or using the GAT framework, rather than the T4 template language used for other generators.

Another aspect where model checking reports differ from other generators is that they often want to check the existence of a certain kind of item, or the number of those items. The DSM tool’s modeling language facilities often support rules to check upper bounds: for example, a Start state may have at most one transition leaving it. Checking lower bounds is however harder, since such a rule could not be checked all the time: adding a Start state to an empty model would be illegal, making it impossible to get started on the model. Lower bound checks are thus often left to checking reports or generators, which the user can run when desired.

Overenthusiastic or bossy metamodelers may be tempted to make such checks compulsory, but that only invites the wrath of modelers when they need to save their work and get out of the office quickly. Such solutions invariably cause more problems than they solve: to pass the checks, the modeler will simply add fake elements to the model. Sometimes those elements will be overlooked the next time, and since no checking report will reveal them, they will remain until your customer finds them for you.

More specialized model verification and validation tasks may be better performed in existing external programs. For example, tools exist for validating that a system, expressed as a state machine, cannot exhibit various kinds of undesirable behavior. A generator could be used to export a state-like model into a format readable by such a tool, and then call the tool. Attempting to analyze the behavior directly with the generator facility would be a poor idea: generator facilities are not renowned for their speed, and these calculations are computation intensive.

11.3.3 Documentation

In DSM, the models form the best documentation of the system: expressed in highlevel terms, yet precise and always up-to-date. The DSM tool also provides the best environment for accessing this documentation, supporting various views, browsers, and queries. However, not everybody will have access to the DSM tool, or be familiar with it, and in any case a strong tradition for paper, Word, or web-based documentation is something that many organizations will find takes time to overcome.

Around the time a DSM solution is being deployed to a pilot group, the first requests for documentation appear. While these are often for documentation of the modeling language, in many tools the modeling language is treated as just another model, and so there is a requirement to generate some traditional documentation format from a model. Certainly by the time the DSM solution is being deployed to a wider group, there will be a desire for producing traditional documentation from the DSM models themselves.

The requirement for modeling language documentation is probably objectively more important for project success. However, making models too available in a traditional documentation format makes management happier, and the resulting sponsorship is an even better determiner for the continuation of the project. We may not like it, but we would be unwise to ignore it: the alternative is to teach the manager to use the DSM tool, but that could be far more dangerous...

Plain text is rarely sufficient to qualify as a traditional documentation format: at least basic character formatting is required. Going beyond the lowest common denominator of plain text poses a problem. It is not sufficient to know that we want the first line to be bold and a larger font, we also need to pick a particular editor or viewer for this formatted or rich text. The choice of tool is influenced by many factors, including:

. who will use the documentation;

. in what environment and form;

. what tools work there or are already present; and . what file formats the tool supports.

The last factor is of course vital for generation, since the most sensible way to output documentation from models is as a plain text file with syntax to express formatting. While it is possible to build generators outputting Microsoft Word’s native binary format, and indeed that would be a fun—if somewhat masochistic— challenge, those electing to output plain text RTF need not feel they are setting the bar too low for themselves. (Sadly!)

Viable candidates for output formats include RTF, Microsoft Office open XML format, SGML, HTML, and LaTeX. These can be used directly, or then fed to an appropriate tool to generate Word DOC files, PDF files, Windows Help files, and so on. Also, taking a higher-level view for a second, we can note that the domain these various languages aim to cover is essentially the same. There are good possibilities to convert documents from one of these formats to another, or produce other similar formats such as DocBook, Texinfo, and groff.

Despite the plethora of options, most projects make a simple choice: if they want to view their documents on a computer, they use HTML; if they want to print them, they use RTF. In both cases, there are three important areas in addition to the text: images, styles, and scripts.

Images Images of the models are of course a key part of the documentation. The generation language must offer facilities for outputting models to separate image files, and making the generated documentation refer to those files. Image files can be either bitmaps, which are generally best for computer viewing, or vector graphics, which are better suited to printing or print-like formats such as PDF.

For bitmap files of models, compression artifacts make JPEG a nonstarter. While the 256 color palette in GIF files is not a problem in most modeling languages, it becomes a limitation if the symbols contain photographic bitmaps or fountain fills. Perhaps the best format is Portable Network Graphics (PNG), since it offers small file size, lossless compression, and a 24-bit color palette. Until recently PNG support in other software such as browsers was weak, but now almost all support the PNG features needed for models.

Up until a few years ago, there was no satisfactory, platform-independent format for vector graphics. PostScript files, and for stand-alone images Encapsulated PostScript, was one possibility, but application support was poor. Most applications could not show the vector content of an EPS file, which would only appear when interpreted by a PostScript printer, but instead only displayed the bitmap thumbnail (if the generator had included one). Windows Metafiles, both WMF and the enhanced EMF, were only supported on Windows. The ancient Macintosh Paint program’s format, PICT, became rather surprisingly one of the better choices. It was relatively simple in its capabilities, but sufficient for the needs of modeling languages, supported 24-bit colors, and could be read in the Macintosh, Windows, and Unix worlds.

The advent of XML saw the introduction in 1998 of two competing XML vector graphics formats, Microsoft’s Vector Markup Language (VML) and Adobe’s Precision Graphics Markup Language (PGML). For a while it looked like the situation would develop into a typical format war, where users are losers. Fortunately both sides, and their supporters, agreed to come together to work on Scalable Vector Graphics (SVG) (how refreshing to have an XML language that does not end in ML!). While Microsoft are still dragging their feet a little in browser support, Internet Explorer can view it with the Adobe plug-in, and Mozilla Firefox and Opera both support SVG natively.

An important part of viewing model images on a computer is the ability for the elements in the models to work as hyperlinks. This allows the user to click on an object in the image, and jump to the part of the documentation specifying that object. In HTML, this can be accomplished by generating an image map; in SVG, the elements themselves can have <a href...> tags. It should even be possible in RTF: invisible drawing objects could be overlaid on a bitmap picture and given hyperlinks with

\hlloc.

Styles The spirit of DSM almost forbids replicating formatting information throughout a generated document. Rather than define each header to be 14 point bold italic Arial with 12 point spacing above and below, the generated document simply contains the fact that this is a level one heading. The actual styling information is moved to a separate file: CSS for HTML, or a template for Word. These files would be handwritten once along with the modeling language, not generated each time.

Splitting the information from its representation in this way also allows one document generator to produce output for multiple purposes. For instance, there may be one CSS file that specifies fonts and layouts suitable for on-screen viewing, and a second CSS file with formatting better suited to hard copy: no hyperlink highlights, no sidebars, and so on.

Not specifying formatting in the generated output also makes it easier to build the generator. A similar effect could of course be obtained with a set of subgenerators to specify the formatting for each style. That tends to lead to rather baroque looking generators that spend more time calling subgenerators than outputting anything useful. The resulting documentation files are also fixed to a specific set of formatting: with a separate stylesheet or template, even old documentation outputs can be viewed with the latest styles.

Scripts Earlier we talked about how more intelligent output formats for plain text make the life of the generator easier. Rather than have to write commands in the generator to produce exactly the required output, the output can consist in part of commands to whatever application will read it. Similar possibilities exist for documentation output.

For output that will be read by a word processor, for example RTF for Word, it is possible to write macros in the Word template that will postprocess the output. One important aim for this in many organizations is to turn the RTF file into an honest-togoodness native Word .DOC file. While there seems no real gain in this, “standard practice” is the Newton’s First Law of software organizations: “every object tends to remain in the same state unless an external force is applied to it.”

More usefully, Word macros can be used to build tables of contents and figures, along with paginating to fit the current default printer. They can also replace links to external image files with embedded copies of those files, making a more coherent single document for distribution and archival. Where formatting cannot be simply expressed as template styles, for example, table formatting with a different background color for alternate rows, a Word macro can easily apply the same Table Autoformat to all tables in the document. While Word’s security model is nowadays by default wary of macros in templates, an organization can cryptographically sign the templates to make them accepted within the organization. Macros can also be used to make the documents live: for example, a button in the document could reopen the model in the modeling tool, either the latest version or the exact same version as in the picture, retrieved from the version control system.

In HTML, scripting is generally less necessary for building the documents. Unlike with Word, the documents tend to be distributed and archived exactly as they were generated. Where necessary, Javascript can of course access any part of the document with Document Object Model (DOM) calls, and most modern browsers support a wide range of operations on the document: parts can be hidden, revealed, replaced, reformatted, and so on. Scripting can also be used to good effect to add dynamic display of information: for instance, a list of an object’s properties can be shown when the mouse hovers over that object in an image.

There are thus many possibilities for beautifying and enhancing your documentation. Since there is however no way to increase the amount of information contained in the models, it is probably wise to think of the models themselves as the real documentation. Rather than try to impress the boss with cool tool tips or professional looking document layout, it may be better to concentrate your efforts on the modeling language and code generators. After all, those will be used far more: nobody ever really reads documentation, do they? And with that rhetorical question, let us move on to another ostensibly human-readable output format: XML.

11.3.4 XML

Generating simple XML is beautiful in its simplicity. Generating watertight XML is an ugly mess. Generating real-world XML is somewhere in between, but sadly the pain increases the more you want to move the level of abstraction of the modeling language above that of the XML files.

Let’s start at the easy end of the scale: imagine we have a domain-specific XML format in which developers currently have towrite files by hand. We create a modeling language based on a simple but often effective pattern: major elements map to graphs, minor elements to objects, and attributes to properties. A generator for any such modeling language might look like Listing 11.6.

Listing 11.6 Simple XML generator.

'<' type '>' newline foreach .() { ' <' type do :() {

' ' type '="' id '"'

}

'/>' newline

}

'</' type '>' newline

The major element is named after the graph type and is opened at the start of the generation and closed at the end. We iterate over each object, outputting an element named after its type, and within that element we iterate over the properties of that object, outputting them as attributes. The generator is not even fragile with respect to changes in the modeling language: we can add object types and change property names, and the generator will still output XML. So far so good.

Element Names But what if the name of an object type consists of two words separated by a space? Then an XML parser reading the result will only parse the first word as the name of the element, and choke when it tries to parse the second word as an attribute name. We thus need to process the names of elements, filtering out any spaces and perhaps other characters that the XML parser would not like. So we take a quick look at the XML specification in the language of our choice... or more likely a rather longer look at the closest alternative (www.w3.org currently lists 16 translations, including Estonian and Interlingua, but “E-Z Read English” seems to be missing).

To cut a long story short, the XML specification for element names is a mess. It may be a very well founded attempt to break the stranglehold of US-ASCII alphanumerics on the ways we are allowed to name things, but from our point of view as we build a generator, it is a mess. If we take just the first character of a name, which has the smallest set of legal characters, it can be a “Letter,” an underscore or a colon. A “Letter” can be a BaseChar or an Ideographic. A BaseChar is any member of over 200 Unicode 2.0 character classes such as [#x0041 – #x005A]: that is the first class, and perhaps more commonly expressed as [A–Z]. At this point the value of a pre-existing XML schema rises appreciably: let the schema determine the object type names, rather than the other way round. This also saves us from the next gotcha: a name may not begin with the characters “xml”, in any mixture of case: those names are reserved.

While accepting an XML schema as given saves us from the task of mapping names in the metamodel to a form acceptable to XML parsers, our problems are only just beginning. After all, we were only planning on having one or two metamodels, but of course many more models. And while we could perhaps have envisaged our metamodelers remembering to only ever give names that were legal in an XML context, that may be somewhat optimistic when it comes to modelers. We therefore come to our next task, outputting property values as XML attribute values.

Attribute Values Even a casual acquaintance with XML will prepare us for the fact that attribute values are quoted. As eternal optimists, we assume that we will thus have to escape or double any quote marks in the value, and then the problem will be solved. After all, if there is only one character out of the whole Unicode gamut that can close an attribute value, surely the parser just reads all characters up to there and then stops?

That, however, would be tantamount to anarchy, and the World Wide Web Consortium, as a standards body, is firmly opposed to anarchy. Standards bodies like to make sure that everybody confirms to a rigidly defined standard format, and the process of making sure of this has a name: normalization. Here is how XML attribute values are normalized, for those who are still innocents:

. Line feeds, carriage returns, and tabs in the value are replaced with spaces.

. Character references such as &#9; (tab) and &#x20AC; (the Euro sign) are converted.

. Entity references such as &amp; (&) and &lt; (<) are converted, and the result of the reference is recursively converted.

Since one of the first problems encountered when using XML to contain values from models is its tendency to mess with white space, we should make sure we know exactly what happens and how to avoid it. The specification states the following— translation is left as an exercise for the reader:

Note that if the unnormalized attribute value contains a character reference to a white space character other than space (#x20), the normalized value contains the referenced character itself (#xD, #xA or #x9). This contrasts with the case where the unnormalized value contains a white space character (not a reference), which is replaced with a space character (#x20) in the normalized value and also contrasts with the case where the unnormalized value contains an entity reference whose replacement text contains a white space character; being recursively processed, the white space character is replaced with a space character (#x20) in the normalized value.

Back to our original question about attribute values: how do we cope with a quote character in a value? The specification allows strings to be quoted either with single quotes or double quotes, but this is little help in our case. Even if we made our generator check each value to see which way to quote it, we would still eventually find a value containing both kinds of quote. Best is to stick with the more commonly found double quotes around the value, and escape any double quotes in the value with the &quot; entity reference.

For the rest, we need to disguise the white space characters the W3C normalization sniper is looking to pick off, and any characters on which the parser would otherwise choke. We also need to check the values for anything that the XML parser is looking to help us out by transforming, such as entity or character references. For example, a property might contain a string “this function maps ampersand to &amp;”. If that were placed directly as an attribute value, the XML parser would read “this function maps ampersand to &”.

Since all character and entity references begin with ampersands, any ampersands in the input must be disguised. We can disguise them as the entity reference &amp; making our example above “map ampersand to &amp;amp;”. While entity references are expanded recursively, the recursion only applies to the replacement text of the entity; otherwise &amp;amp; would first become &amp; and then just & again.

Since all our other disguises will also use the & character, we had better disguise any real & first as character references. For some reason, the convention seems to be to use a decimal character reference for &, hexadecimal for white space, and named

TABLE 11.1 Escaping Special Characters in XML

|  |  |  |  |
| --- | --- | --- | --- |
| Input |  | Output |  |
| Entity Reference | Decimal Character Reference | Hexadecimal Character Reference |
| & | &amp; | &#38; | &#x26; |
| < | &lt; | &#60; | &#x3C; |
| > | &gt; | &#62; | &#x3E; |
| 00 | &quot; | &#34; | &#x22; |
| [tab] |  | &#9; | &#x9; |
| [line feed] |  | &#10; | &#xA; |
| [carriage return] |  | &#13; | &#xD; |

entity references for the other characters. These are thus given in bold in Table 11.1, but any of the given forms may be used.

Text Elements Another way to store values in an XML file is of course as text between an element start tag and end tag: <tag>like this</tag>. In simple text sections like this the above mappings are also valid, although there is no need to map quote marks. White space is however at risk here too: how much depends on a variety of factors. While parsers are not allowed to attack it, applications are expected by default to normalize it. They can normalize it in any way that pleases them, but something similar to the above attribute value normalization is likely. Adding an attribute xml:space=“preserve” is one way to tell applications to keep their hands off text in that element and its subelements. Even then, your spaces are not necessarily safe: for example, the SVG specification states that with xml:space=“preserve” all white space characters are mapped to spaces, and leading and trailing white space is removed. And remember that this is just the specification: individual SVG applications may not necessarily follow it perfectly. Ah, the joys of having a simple standard like XML that guarantees interoperability!

XML also has a way to mark text sections so they would ostensibly be completely spared the attentionsof parsers’ normalizers and transformations.Thetext section must be enclosed in a CDATA section by surrounding it with the underlined text below:

<tag><![CDATA[likethis]]></tag>

Within a CDATA section, & and < may occur as normal: there is no need to escape them, nor indeed are any of the normal character or entity references even recognised. The only character sequence that cannot occur inside a CDATA section is ]]>. One way to get around this problem is to split input containing that character sequence into two CDATA sections. “You cannot use ]]> inside here” becomes

<![CDATA[Youcannotuse]]]]><![CDATA[>insidehere]]>

CDATA sections unfortunately also still mangle carriage returns: that normalization happens before the document is even parsed. Any carriage return or “carriage return + line feed” pair is mapped to a single line feed character. Sadly, there seems to be no solution here, since CDATA sections cannot protect their line breaks from the normalizer by disguising them as numeric character references.

Encoding Finally, as with any file output, there is the problem of character encoding. If your XML file will only be processed on the computer where it was generated, you may hope things would just work, and indeed in this case they just might. If you hope that your file would only ever contain good honest 7-bit ASCII characters, you will almost certainly be disappointed: aside from exotic foreign characters and exciting new currency symbols, somebody is bound to paste some “smart quotes” from Word into a model, opening Pandora’s box.

No single choice of encoding can be recommended over all others, because your needs will vary according to the language used in text in models, the operating system and byte order on the computers used to generate XML from the models and to parse that XML, and the tools used to transmit, process, and parse that XML. The only encodings that all XML parsers must recognize, however, are UTF-8 and UTF16, so if you have no strong reasons to choose something else you should try one of those:

<?xmlversion="1.0"encoding="UTF-8"?>

11.3.5 Flow Machine

Now we have looked at plain text, formatted text, and structured text, we can move on to the more meaty subject of generating program code. The simplest kind of program is a linear sequence of instructions, so let us begin there. While such simple programs are rarely a major part of a DSM solution, almost all DSM solutions generating programs will include command sequences in some form.

Sequential In Chapter 7, we saw an example of a generator for simple sequential code in a microcontroller application. The modeling language showed the commands as objects, chained together into sequences with relationships. Perhaps surprisingly, building a generator for such cases requires a little thought.

Most generators work iteratively, often in a tree structure determined ahead of time by the modeling language rather than the model. An iterative approach will not work here, at least not in a DSL-based crawler generator. The smart iteration operators in such a generator are too specialized: what we would really want is a lower-level dowhile block. This would print out the command for the current object, traverse to the next object, and repeat as long as that next object existed. In an API-based generator, it would look something like Listing 11.7.

Listing 11.7 API-based generator for sequential code.

do {

System.out.println(currObj.Command()); currObj = currObj.nextObject(); } while (currObj != null)

With a crawler we can use a recursive approach, as shown in Listing 11.8. We output the command for the current object, then traverse any outgoing relationship to the next object and recurse.

Listing 11.8 Crawler-based generator for sequential code.

Report 'handleObject'

:Command /\* output for this object \*/ do ~From~To.() /\* traverse to next object(s) \*/

{ subreport 'handleObject' run /\* recurse \*/

}

These approaches will work providing there are no cycles: in simple languages like this, cycles may well be forbidden by the modeling language anyway. A benefit of this simplicity is the minimal program size and execution time of the resulting code: it really is just a sequential list of commands.

Conditionsand Jumps To be useful in practice, most programs must have more than sequential execution: we need conditions and jumps. At its simplest, this means the modeling language equivalent of IF and GOTO—and indeed if we are generating assembly language, that may be all we can use in the output.

In models, IF can be represented either as an object, from which True and False relationships exit, or else as an n-ary relationship. It can also be extended to a switch– case or ELSEIF structure, with one exiting relationship or role for each choice. If the control paths rejoin after being divided by the IF, as in the top half of Fig. 11.1, a naı¨ve recursive generator will produce poor results. The resulting code will work, but will duplicate the rest of the program for both branches. In cases more complicated than the figure, even maintaining a list of visited objects will not be sufficient to avoid problems in a recursive generator.

A recursive generator can be made to work with IF structures by changing the modeling language. Rather than requiring both control paths to rejoin to the main control path, we can model the branches in an IF structure as subordinate to the IF statement. An example of this is shown in the bottom half of Fig. 11.1. The content of the “Foreign?” conditional statement includes its Yes and No branches, each of which is followed by its own new recursion. The next statement after the conditional is the Mail action, which the main sequential program recurses to. The resulting structure is thus more similar to how a compiler might parse the program: at the top level there are

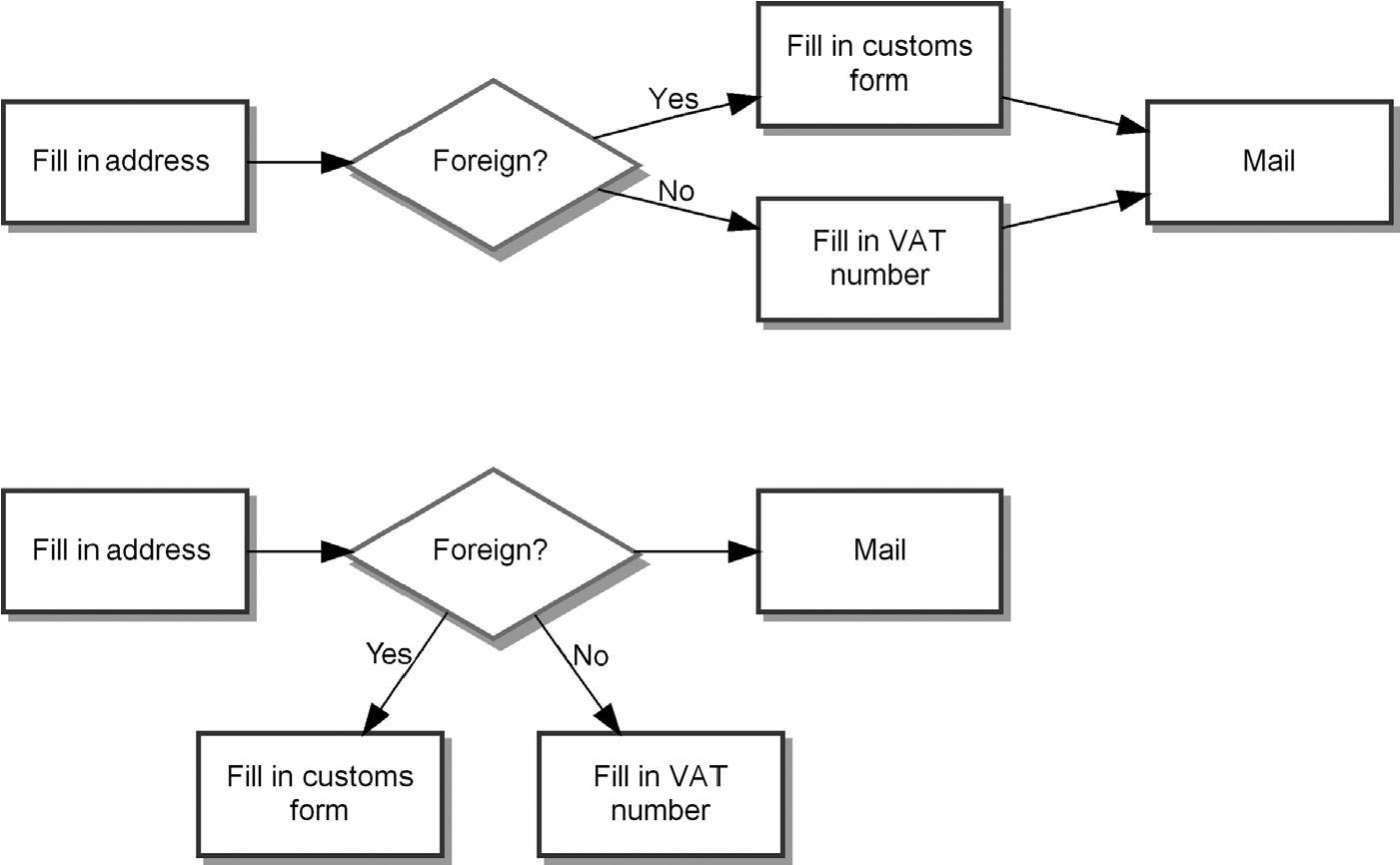


FIGURE 11.1 Rejoining versus subordinate condition

three instructions, the second of which contains its own subtree. A generator might look something like Listing 11.9.

Listing 11.9 Generating subordinate conditionals.

Report 'handleObject'

if type = 'IF' then /\* conditional object \*/

'IF' :Condition 'THEN'

do ~From>Yes~To.() /\* new recursion down Yes path \*/

{

subreport 'handleObject' run

} 'ELSE'

do ~From>No~To.() /\* new recursion down No path \*/

{ subreport 'handleObject' run

} 'END IF' else

:Command /\* simple object \*/ endif

do ~From~To.() /\* main flow recursion \*/

{

subreport 'handleObject' run

}

As we will see below, we are not forced to change the modeling language to cope with IF: the other option is to change the generator structure. Before we look at that, we ought to consider GOTO. This may seem a terrible regression: surely DSM is meant to be about best practices, yet here we are talking about a structure that has been frowned on for nearly 40 years. Why then is GOTO not considered harmful in modeling languages? One main reason is that the path of control is far clearer in a graphical model: we can see a line from each GOTO call site to the code it calls. Contrast this with textual programming languages, where the reader may have to scan a large text for labels and compare each label to the GOTO argument. Interestingly, this provides an argument for the use of line numbers in code, both on each line and as the GOTO argument: while the semantics of the call become less clear, the target of the call is easier to find. Even that only helps in showing where the call goes to; graphical models also show all places that call a piece of code.

There is also a second reason why GOTO is not as bad in models as in code. While we can use GOTO to make spaghetti models as well as spaghetti code, in the models the spaghetti is instantly visible. The instant readability of the control paths warns the developer early if things are going too far.

A simple sequential flow is itself a kind of GOTO: in a program, the program counter is set to point to the next statement, just as it is set with a new value in a GOTO command. In a model, the sequential flow is shown by an arrow, and the same arrow can be used for a GOTO. The use of a GOTO is thus only really of interest in models when the target is something other than the next object: in other words, when the target is an object that can already be reached by another path.

Where paths join in this way, a recursive generator will encounter the same problem as in IF statements above. In this case, the solution is simpler and was already seen in Chapter 7. We can make a Label object connected to the start of the chain of commands we want to jump to, and connect our GOTO to that object. Since GOTO can only be connected to a Label object, we do not need to follow GOTOs recursively. If we also forbid normal sequential flow to Labels, we can generate the program by starting at all Labels that have no incoming relationships, and generating recursively from each. Where we would have liked a Label to be part of the sequential flow, we simply move it out and make that sequence call it via a GOTO.

Even this approach will only take us so far, just as it only took early programmers so far. As programs grew, they found that they needed more advanced constructs. However, such growth is by no means a certainty in DSM, and may be a symptom of trying to build a modeling language that mimics program language constructs, rather than focusing on the problem domain. If the growth does occur, the next logical step is GOSUB: GOTO with a return back to the same place. And following hard on GOSUB’s heels are its more familiar and powerful implementations: functions.

Functions The problems of sequential generation, whether iterative or recursive, are largely caused by the same object being reachable by more than one path. While maintaining some sort of collection of already generated objects may help in some cases, it does not solve the difficulty of finding where two complicated branches of an IF statement merge back into the main flow. If we want our modeling language to remain more like a flow chart than an abstract syntax tree, we need our generator to move beyond sequential lines of code.

The simplest approach is to map each object into its own function. The function will take care of the actual actions of that object, plus the specification of which function to call next, possibly depending on a condition. In a Visual Programming Language (VPL), this approach would generate large amounts of functions, each with only two or three simple lines of code. In a Domain-Specific Modeling language, one object represents more than just one atomic code operation, so the functions will each accomplish far more. Of course, we will often be able to spot patterns within the function bodies, and abstract those out into the domain framework, leaving just a simple parameterized component invocation behind. While the remaining function bodies may be only two or three lines long, they clearly accomplish far more than the atomic operations found in a generic VPL.

In simpler function-based output, each function explicitly calls another function at the end. In most applications this will be no problem, but it is worth noting that each call will add a new entry on the call stack. If stack space or memory is limited, this can be a problem. In particular, if there are objects connected in a cycle, going round the cycle will continually add new entries on the call stack, eventually running out of memory.

The solution to this problem is to implement a kind of tail recursion. Tail recursion may be familiar from functional languages, where many algorithms make use of recursive function calls. Each invocation of the function at a new level of recursion would normally add a new entry to the call stack. Only at the end of the algorithm would all the calls unwind, with each simply returning its value to its caller. To prevent stack overflow, functional programmers noticed that there is actually no need to maintain all the function invocations on the stack. The last step of the function is often just to return the value of calling itself recursively. In that case, it is enough to maintain one invocation of the function on the top of the stack, and each new call simply overwrites that copy. At the end of the algorithm, the deepest invocation thus returns its value directly to the caller of the first level of invocation.

While our function calls are not recursive, we can note that the last step in each function is to call the next function. Instead of making that call from within the function, we can have the function simply return the next function to be called. An outermost loop can then call the function, receive as a return value the next function to be called, call that, and so on. The call stack only ever has one element in it, and we can happily have long chains or even cycles with no worries of running out of space.

This is the approach taken in the Python example in Chapter 8. The main() function there calls the current function, receives the next function back as the return value, and calls that. Python is a sufficiently powerful language that functions are treated as first class objects, which can happily be passed around and called.

Other languages vary in their support for dynamic method invocation. Smalltalk, Lisp, and Ruby predictably have no problems, but even the lowly BASIC-like OPL used for end-user programming in Symbian is able to call functions whose name is held in a string variable. While not quite as powerful as having functions as first class objects, this is sufficient for our needs. C is happy to offer function pointers, but some more recent languages have more trouble. The Standard Edition of Java can cope via its invoke(), but this requires a fair bit more code than OPL or Smalltalk’s perform:. Java on mobile phones, J2ME MIDP, unfortunately lacks the reflection libraries that contain invoke(). C# has Type.InvokeMember, which is rather wordy with a receiver and five arguments, but basically similar to Java’s invoke(). While C++ has poor reflection capabilities, it is possible to drop down to C function pointers or then use member function pointers.

Overall, then, almost all languages support functions passing back the next function to call, rather than calling it directly. As the code to make the call will only appear once in the outermost loop of the program, even the frightening number of arguments of C#’s InvokeMember, or the equally unpleasant number of exceptions thrown by Java’s invoke(), should not put us off.

11.3.6 State Machine

State machines are among the earliest theories of computation, dating back to Turing (1937), and even the most prevalent “modern” forms, Harel’s statecharts (1987) and SDL, are over 20 years old. They share many similarities with flow machines, and much of the discussion above is thus relevant to generation of state machines. There are however some important differences.

In normal flowcharts, each object type can have its own kinds of exiting roles. For instance, IF is exited via a True or False role, or a switch case is exited via any of the various case values or the default case. Many objects exit on a simple “done my stuff, time to move on” basis. These decisions are thus dependent on the object type and tend to be based on the internal state of the program.

In contrast, state machines have a global concept of what can trigger a transition from one state to another. Typically this is an external event, although it can also be an event caused internally. States thus operate on a more laid back “done my stuff, now I’ll just hang around here” basis, waiting for something to kick them out of their reverie. While states thus lead a more relaxed life than flow machines, not everything is easier. Whereas in a flow machine each object could decide itself when it was done, and where to go next, in a hierarchical state machine an event in a toplevel model can cause us to jump up out of a state in a lower-level model, even though that state specified nothing about the event. When the man at the top says jump, you jump.

Actions in a state machine are most often specified along transitions, although they can also be specified in a state, for execution on entering or leaving that state. In a DSM language, some features of a state can be specified as properties of the state, as opposed to actions. Recall in the Watch example, each state specified whether a certain time unit was flashing while we were in that state, and which display function to use while we were in that state. Other actions—icons turning on or off, time calculations, setting alarms—were specified along transitions.

State machines are considerably more complicated than the flow machines above, especially if we look at full Harel statecharts. Compared to those, the watch example had no conditions on transitions, no state enter and exit actions, and no state history, and event transitions in a higher model did not affect lower models. This is a common state of affairs in a DSM language: we implement only a fairly minimal subset of all the features a full theoretical modeling language might have. This helps considerably when building and using the modeling language, but also when building the generator and domain framework.

Switch–Case In embedded applications, the most commonly seen code for state machines consists of nested switch–case statements. The first switch is based on what state the application is currently in, and the second is based on what event has occurred. As we saw in Chapter 9, generating this kind of code from a state machine is simple. For ease of understanding, we can omit some details here and assume that EntryAction and ExitAction are properties of State objects, while Condition, Event, and Action are properties of Transition relationships—all expressed directly as legal C code. This will give us the basic generic generator in Listing 11.10, shown here in a hypothetical template-based crawler language.

Listing 11.10 Simple C generator for state machines.

switch (state) { <% do .State %> case <%=id%>: switch (event) { <% do ~From>Transition %> case <%=:Event%>:

if (<%=:Condition%>) {

<%=~From.State:ExitAction%>;

<%=:Action%>;

<%=~To.State:EntryAction%>; state = <%=~To.State%>;

} break; <% enddo %> default: break; } <% enddo %> default: break; }

We start the switch statement, then iterate over each state. For each state we generate a case statement and an inner switch statement. Inside that we iterate over each transition from that state, generating a case statement for each event. Within this innermost case we generate an if statement for the condition, and inside it the exit action for the previous state, the action along the transition, the entry action for the new state, and an assignment that sets the new state as the current state.

Transition Table A second way of implementing state machines is to represent them as data rather than code. This approach is seen more often in object-oriented languages, whereas the switch–case approach is most common in procedural languages like C. More complex state machine modeling languages also tend to be better represented as transition tables.

As we saw in the Watch example in Chapter 9, each model can map to a new subclass of a state machine class. The state machine class provides a variable to hold a table that maps each state to the transitions for that state, and the subclass constructor fills in that variable for this model. The Watch example also had other tables that mapped states to their decompositions, display functions, and blinking information, but these would often be better represented in similar variables in individual State objects. The information recorded for each state and transition is thus domainspecific, in contrast with the not infrequent writings proposing “the ultimate state machine implementation in Java/C/Snobol.”

While the details vary, the main task of the state machine superclass is to receive events and push them through the state machine. It has a variable for the current state, or possibly a stack if state history is to be implemented. Looking up the current state in the transition table returns a table that maps events to transitions. If the event received is in that table, the transition it points to is considered for evaluation. Any condition on the transition is evaluated, and if that passes, the current state’s exit action, the transition action, and the new state’s entry action are performed. Finally, the current state variable or stack is updated to reflect the new current state.

In the Watch example, the states and button events were represented simply as Strings and the transition tables were Hashtables. It is also possible to make State objects, whose contents will then closely reflect the properties of the State objects in the model. In a procedural language implementation we could move in the opposite direction, and use integers to represent states, with arrays replacing the Java Hashtables. While an attempt to write a generic state machine in C would have problems with such an approach because the size of the arrays could not be known, in our case the number of states in each model is known at generation time. By judicious use of enum variables, the implementation could thus be simultaneously easy to read, fast to execute, and small in code size.

The transition guard condition and the various actions are of course rather harder to map into data. Important here is to remember that in the models, these should not be represented as code fragments, but as domain concepts and actions. One useful approach is that for every kind of action, there is a corresponding modeling language concept—an object, role or relationship, depending on how the state machine is modeled. Still, these actions may often be generated as direct inline code for readability, so the problem remains.

How then do we include pieces of code in a data structure? Some languages such as Smalltalk offer simple support for code as data, and in other languages it can often be accomplished with a little work, for example, in Java via inner classes. If that is not possible, the discussion on dynamic method invocation at the end of 11.3.5 above is worth revisiting: we can implement each piece of code as its own function, and just supply a reference to the function in the data structure. Where even that is not possible, we can use the simple approach seen in the Java Watch implementation: make one big function that contains all the pieces of code in the model as cases in a switch or IF– ELSEIF statement.

11.3.7 Integrating Handwritten Code

Even a passing acquaintance with code generators will have taught you the cardinal rule: never edit generated code. If you need different code than was generated, make the necessary changes to the model. If you cannot accomplish what you want just by editing the model, look at improving the generator or modeling language to handle this and similar cases. If the code needed for this particular case really does seem to be a one-off, the modeler will need to write the code by hand. Even then, the changes cannot be made ad hoc to the generated file, or they will simply be lost when regenerating after subsequent changes to the model.

It is important here to differentiate between handwritten code that will be the same for all (or many) applications in the domain, and code that must be written by hand for a single part of a single application. The former forms the domain framework, which is written by the DSM solution team, and will be covered in the next chapter. The latter is written by the modeler of the application in question, and it is on such handwritten application code that we focus here.

Handwritten application code can be integrated in three ways:

. Protected regions

. Handwritten code in models

. Files referenced by models

Protected Regions A protected region is a section of the generated file that is known by the generator to be possibly edited by hand. When code is regenerated, the generator reads that section of the file on disk. If it has been edited, the contents of the region on disk are preserved, overriding that section of the newly generated version of the file.

To make this possible, the regions are generally delimited by specially formatted comments. The generator recognizes such comments and can parse information from them to link that region with the corresponding region in the new output. The comments also contain a checksum of the original generated code, so the generator can check if the region has been edited. Other solutions are of course possible: for example, for the DSM tool to remember the line numbers and checksums of the regions without adding comments to the files. This is however prone to problems: for example, if files are replaced so the file on disk is no longer the previous generated file whose information has been remembered by the tool. Region comments also make the task of editing the files easier, by clearly showing which regions are protected.

There are a number of other details still to be covered, but we shall skip them here because protected regions cause more problems than they solve. They force DSM users to work on the models, the generated code, and the hand-edited code, and mix the latter two up in a troublesome way. Both models and code files need to be versioned, so the same information is recorded twice. Although protected regions may feel familiar from round-trip UML tools, there are better ways of achieving the same results.

Handwritten Code in Models Rather than adding handwritten code to a generated file, the code can be added to the model. Obviously, this should be avoided as far as possible, but when handwritten application code is a necessity, putting it in the models allows you to keep a single source containing all necessary information. The details vary according to the need and the modeling language, but one solution is to add a “Hack” object type to the modeling language. The object contains a free form text field, into which the modeler can enter one or a few lines of code. The generator for these objects simply outputs their code at the appropriate places of the output file, as determined by the positions and relationships of the Hack objects in the model.

Having a single source can be good, in particular if the Hack objects are short enough to be read as part of the model without cluttering up the display. If they become too large or numerous, they will tend to reduce the efficiency of using the DSM language. Also, code longer than a line or two would benefit from being entered in an IDE, rather than a simple text box. It is thus time to look at the final way of integrating code and models.

Files Referenced by Models The obvious next step from code in models is to move the code to its own files and have the models refer to the files. This promotes the “Hack” object to an “External Function” object, with a simple string representing the file name. Each piece of code can be in its own file, and that file can be edited in an IDE—at best along with the current generated code, easing any necessary integration with that code. Of course, it is best to keep such coupling to a minimum: the generated code is allowed to expect certain things of the handwritten code, but generally not the reverse.

In the simplest cases, the generator can output include or import statements in the generated code to refer to the handwritten code. The generator should also produce a list of the referred files, and check that they exist. If there are more than a few such files per model, it may be easier to move the pieces of code from each being a file to each being a function, with all functions from one model contained in a single file.

At this point, we have moved into the territory of programming in general. Tactics to allow further expansion of this idea include automatically generating placeholder functions, which can be overridden by handwritten code in a concrete subclass. Moving further in this direction, however, takes us out of the domain of DSM and into the kinds of solutions we see in older modeling tools. As those tools have failed to raise developer productivity, if we find ourselves adopting their tactics we would do well to step back and reconsider our DSM solution. Handwritten application code is not quite the epitome of evil, but neither is it by any means a necessary evil.

11.4 GENERATOR STRUCTURE

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—but not all—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.

In this section we will look at various ways to structure generators. Each individual way can be taken and used on its own, or in any combination with the others. We have however found that building a hierarchy of generators top-down in the order specified here works particularly well.

A top-level “autobuild” generator is thus visible to the user, and is responsible for building the resulting application for testing. It calls generators defined on the various modeling languages, 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.

11.4.1 Autobuild

The Agile movement has grasped the importance of continuous integration via automated builds, even though their second tenet is to favor “individuals and interaction over processes and tools.” The truth is, every application must be compiled and linked to be any use (substitute compile and link with appropriate terms for your language!). If you do not automate this process, developers will be forced to carry it out manually every time. In addition to being error prone and hard to reproduce, this is also boring, leading to developers not doing it when they should.

Our experience in DSM is that a working autobuild provides far more value than could be imagined. In particular, automatically running the generated application provides a new level of abstraction that has to be experienced to be believed. There is a real power in going directly from modeling to seeing the full application running—not a prototype or simulation, but the actual application. As the models are on a high level of abstraction, often close to the concepts of the end user of the application, you really have the feeling of staying focused on what you actually want in the application, rather than the details of how to implement it.

As the saying has it, though, you can’t make an omelette without breaking some eggs, so for others to be able to ignore the details of how to implement their programs, we need to look here at the details of how to implement such support. But hang on a minute, isn’t there some clever meta trick we could apply here, so we can ignore the detailsofthistask?Funnilyenough,thereis,andyou’llbeforgivenyourlackofsurprise when we tell you it’s using a domain-specific language. In the domain of building software out of an assorted collection of files, one simple word says it all: make.

The make tool allows you to specify what files your finished application is dependent on and what command is necessary to process those files into that application. There may be several stages and intermediate files, with different commands necessary at each stage. For instance, an executable is built by a linker from object files, but the object files themselves must be built with a compiler from source files. With a little work, it is possible to make a makefile (the input to the make tool) that simply lists the source files, the general command to compile source files, and the general command to link object files.

When writing software by hand, keeping such a make file up-to-date is just a matter of remembering to add a file name each time you create a new source file. When building software from models, we definitely do not want the modeler to have to know what source files we are creating, and remember to update a makefile. Since the generator is creating the source files—apart from some framework files that are fixed regardless of the models—it is easy enough to generate a list of the files for the make program. If your generator language supports it, the easiest way is to append the name of each source file to a variable or temporary file as you start generating it. After generating the source files, you can generate the make file, including the contents of that variable or temporary file at the appropriate place.

The alternative to using make, or its equivalents such as nmake and successors such as ant, is to generate and execute the command lines for the build commands. You can either execute the build commands directly from the generator or generate them into a batch file and execute that. In general, the latter approach is easier to work with, since the batch file also forms a record of exactly what commands were executed.

You also want to make sure you get a record of any errors that occurred, especially while you are still working on the generators. One good way is to redirect error output to a file, and open that file in a text editor if there were errors. Below is an example for Java on Windows NT/2000/XP. The first line runs the Java compiler, redirecting error output to an errors.txt file. The second line checks if the compiler returned an error code, and if so opens the errors.txt file and exits. The /b exits the batch file, as opposed to the whole command shell, and the 1 makes the batch file return an error code, so a calling program knows there was a problem.

javac\*.java2>errors.txt iferrorlevel1starterrors.txt&exit/b1

The main difference between using make and directly calling build commands is in how platform dependent the build process is. Make largely insulates you from changes in build platform, and ant does so almost completely; a batch file or shell script will however only run on a Microsoft or Unix OS, respectively, and will quite probably be dependent on a particular version of that OS. If you find yourself having to write command scripts on several platforms a good resource is www.ss64.com, which documents the commands for Windows, Linux, and Mac OS X.

One main benefit of make is often lost when moving from writing source files by hand to generating them. Make is able to see which files have changed since the application was last built, and only update those intermediate files that are dependent on the changed files. Generators normally regenerate all files, since they cannot tell which models have changed (in particular where the same object is reused in multiple models). This means all files look new to make, and all will be compiled, sometimes taking much longer than would strictly be necessary. Avoiding this problem is simple, and the better DSM tools already support it automatically: if a generated file is identical to the file of the same name already on disk, leave that file untouched. If your DSM tool does not support this, it is possible to create the same effect by generating source file contents with a temporary extension, comparing them to the previous source file, and only copying them over the previous version if there is a difference.

While the bulk of work in an autobuild generator will be in creating and executing the makefile or build script, we must not forget the surrounding work. First, the autobuild generator must run the other generators to actually produce the code, before executing the build: the following sections will cover this. Finally, after a successful build the autobuild should open the resulting program. If the program is a desktop application intended to run on the same platform it is developed on, it can simply be started. If it is targeted for a different platform, it can often be opened in an emulator or simulator. As these are often slow to start up, it may be worth investigating if the emulator itself can stay open, with each autobuild simply updating the application in the emulator. If the generated program does not have a user interface, and hence there is no point opening it, the final step of autobuild could also usefully be to run automated tests against the program and display the results.

11.4.2 Generator per Modeling Language

A full DSM solution for a given domain will normally have more than one modeling language. The kinds of information needed to describe an application will be divided over a few modeling languages. These modeling languages, and the models madewith them, should exhibit high cohesion and low coupling. If this is the case, the information needed by the generator at a given stage of generation will generally be found from the same model the generator is currently in. A generator will not need to hop back and forth much between different models in different modeling languages: if that were the case, the modeler would most likely have to perform similar mental acrobatics in creating and reading the models.

The partition of the domain into modeling languages tends to offer a good basis for partitioning the generator. Following the same partitioning closely also brings the benefits of simplicity to the metamodeler. Associating each generator with one modeling language gives an encapsulation of data and behavior similar to that found in object-oriented programming. The modeling language defines the data structure, and the generator is the code that operates on the data.

Autobuild thus often calls a top-level generator per modeling language. The top model may generate significant code itself, in which case it will have a top-level generator in addition to the autobuild generator. Alternatively, the top model may be more like a set of links to the real models, with little other content of its own. In that case autobuild will simply iterate over the links, calling the appropriate top-level generator for each target model.

Each model may contain further links to other models. If these submodels are in the same modeling language, the generation for them will most likely be a recursive call to the top-level generator, but now on that submodel. If the submodels are in a different modeling language, they are somewhat more likely to be including information by reference rather than containment. The generator may then simply pick up the information it needs, or generate a reference, without needing to process the full content of those submodels. The full content will be generated via some other path from the top-level models.

In some cases, the information partition that is best from the point of view of the problem domain may be at odds with that required by the implementation domain. The implementation language or frameworks may require a certain set of information in a file, but that information might be spread over several models in various modeling languages. In that case, it might make more sense to have autobuild call a generator per file or per file type, and have that generator call partial generators for each modeling language it needs to visit for information. Generally, though, the pattern is that a generator per modeling language will contain a number of subgenerators, one for each file type to be generated.

11.4.3 Generator per File Type

Each model may generate more than one file: as mentioned above, there may indeed be a many-to-many relationship between models and files. This is problematic for generators based on templates or the model visitor pattern. While generation into multiple files may be possible, you tend to find yourself working against the tool rather than with it. API-based generators can of course cope, and stream-based crawler generators come into their own.

There can be several different reasons why more than one file may be generated from a single model. In some languages, the implementation may require multiple files, for example, a .c and .h file for C programs. In other languages, standard practice may encourage them, for example, a Form1.cs C# program may have a separate partial class for its UI definition, Form1.Designer.cs. These cases are predictable, with one model always giving a pair of files. More complicated cases are also common, for instance because the modeling language represents information in a significantly condensed form.

One DSM model may simply contain the same information content as several source code files. A fair part of this reduction is removal of redundancy, which can be split into two parts. First, each source code file contains significant similarities to other files of the same type. Constructors, accessors, database links, and serialization are often made up of the same few lines of code, with the only variation being the name of the piece of data in question. Second, many of today’s systems require multiple types of code files for one conceptual entity: a C# interface, data service class, database table definition, database stored procedures, and so on. When these are written by hand, the names and other details of the attributes of the entity must be duplicated across all these files, but in the model each is represented only once.

In addition to code files, or code-like files such as database table definitions, there are also often various other files required as output from a model. For example, the Symbian S60 platform can require as many as 27 files for a simple “Hello world” application: menu and UI definition files, localization files, application descriptor files, icon lists, and so on. Some of these can be generated directly from the information in the top-level model of the application; others such as the localization files must recurse through all the models with UI information. This is easy for a crawler or API-based generator: they can simply open a new stream for each file and walk the structure of the models to visit all the required information. Template-based or strict model visitor generators may well require some extra work or hacks to produce the required output, as noted above.

11.4.4 Generator per Object Type

When generating from a single model, the form of the output required normally varies according to the type of the object in question. This is the basis of the model visitor pattern, which says that each object type should have its own generator. Where the strict model visitor implementations fall down is when information from the same object is required in several different places in different forms. This is no problem for other generators, but they would still do well to make a clear connection between the object type and the generators for it.

In most non-API generators, object types themselves do not have their own generators. Instead, the generators are stored globally or with each modeling language. In this case, the subgenerators for each object type can be named after the object type. Where the object type requires different output for different parts of the generation, the names can be formed by prefixing the kind of output required to the name of the object type, for example, Action\_Alarm and Action\_Icon for the Alarm and Icon types’ actions in the Watch example.

This order is generally better than the reverse order—which would be more familiar from object-oriented programming—because there tend to be more similarities between all of the Action generators than all of the Alarm generators. Unlike object-oriented classes, the object types have no notion of hiding the implementation of how they store their information. Access to the properties of the object is direct, and access to relationships and subgraphs of the objects is generic, working the same regardless of the type of the object.

Some DSM tools equate models and their elements closely with object-oriented classes:anyinformationaccessmustbecarriedoutfollowingtherulesofstaticallytyped languages.Asidefromtheextraworkthisentailswhenwritinggenerators,suchtoolsalso often choose to make access to relationships happen through objects: an object stores its own relationships. The problem of this becomes apparent when you want to reuse the same object in a different model: you may well want it to have different relationships there, but the generator cannot tell which relationships belong to which model.

A more useful approach is to recognize that information about which relationships connect which objects belongs to the model. Since while generating we are always within a particular model at any given time, we can always follow the relationships of an object in that model. This also allows the generator language and generators to be simpler: there is one global mechanism for following relationships, specifying the type of the relationship as an argument, rather than having a separate message for each kind of relationship for each object.

It is also worthwhile noting that in some cases, relationships and even roles may have their own subgenerators. An example of this was in the Watch Application models in Chapter 9: the typing of actions was via the relationship type, and the details of the action were contained in the relationship or its roles. For instance, a Roll relationship connected to a Time Variable, and a Boolean property of the relationship specified whether the action incremented or decremented the value of the Variable. Similarly, a Set relationship could connect to the same Time Variable and, say, an object representing the system time: this would set thevalue of the Variable to be the current system time. Since the types of the objects involved in both cases were the same, and the type of the relationship was what determined the semantics of the action, it made sense in this case to have a generator per relationship type.

11.4.5 Parallel Generators per Programming Language

The preceding types of generator cover the output of a full working application, containing all the information from the models, marshaled into the required files and formats for the given platform, domain framework, and compiler. In some cases, even this is not enough: we may want to be able to generate the same application in more than one way. The most common need for this is where we want to generate the same application for more than one platform. This however is normally fairly easy to accomplish by using a different domain framework for each platform, but maintaining the same interface between the generated code and the framework. In many cases, the generated code can remain the same and only the framework changes; in others, one set of generators may still suffice, including a few conditional sections that depend on the platform for which we are generating.

A more difficult task presents itself if for some reason we must generate the same application, but in a different programming language. The information content of the generated output will be the same, but the distribution over files, ordering of the information in the files, and syntax of those files will all change. In some cases, fortunately rare, the best architecture for the application will differ so greatly between the two languages that the code itself will be largely unrecognizable.

None of these of course presents any problems to the modelers: they simply build one model and choose which generator to use. The hard work of building two parallel generators is left to the metamodeler: once again, the one suffers for the sake of the many. The root cause of needing to output two different languages is of course worth investigating. Unfortunately, even where there seems little rational call for it, greater forces may be at work in the form of senior management and stakeholders. If so, console yourself with the thought that you can at least blow their socks off by using DSM to achieve the same net result of the holy grail of automatic translation between languages!

For model visitors and templates there will once again be the problem of how to allow multiple parallel generators for a given object type or file. Crawlers and APIbased generators will simply duplicate the relevant generators. If the architectures and languages are similar, some higher level generators may be reusable between both languages. If they simply follow model structures and call subgenerators, as opposed to actually outputting code themselves, they can simply be parameterized to know which language version of the subgenerators to call.

The more different the architectures are, the less benefit there is to be gained from attempting to keep the generators similar: at some point it becomes easier to simply write a new set of generators from scratch for the second language. Whatever solution is chosen, there should be some process in place to ensure that changes made later for one language’s generators are propagated to the other language’s generators when necessary. The chance of there being any way to automate these changes is as slim as that of automatic translation becoming 100% reliable, but this doubtless will not stop some from trying. Before you embark on that route and rediscover the only workable solution—having some intermediate form from which you can generate both languages—stop for a second and remember that you already have precisely such an intermediate form: the models.

Automatic Translation and Intermediate Formats There have been attempts in some DSM tools to force the use of such an intermediate form between the models and the code. The background to such attempts is however not from the world of DSM, but rather from UML. In UML, the domain is object-oriented languages, and there is no expectation that working applications will be generated from models. Instead, it is sufficient for the UML tool to generate the class and function headers. As there is no problem domain semantics in the models, and hence no knowledge of what the code should actually do, the generation simply maps a graphical description of object-oriented classes into a textual format. At that level, object-oriented languages are similar enough that simple mappings can be made, and where the mapping is less than ideal, it does not matter anyway, since nobody is going to execute the code.

Coming from such a background, where all modeling languages are UML and all programming languages are object-oriented, it is tempting to envisage DSM code generation as being first a transformation into UML, then a transformation per language from UML into code. There is generally an implicit assumption at this stage that there need only be one transformation written from UML to Java, say, and all DSM languages could use that same transformation. Had that been true, then UML would have provided what its hype implied: full applications from models.

Unfortunately, the model to code transformations for two different languages, even closely-related object-oriented languages, are rarely so similar that they could be considered the same transformation with just a few differences in syntax. Give the same problem description to a Java programmer and a C++ programmer, and the resulting programs will be distinctly different. Good programmers will take advantage of the features of the language—and have to work around its foibles and shortcomings. Automatic translation between object-oriented languages is a nice pipe dream, but not a practical approach; the same is true for universal intermediate formats.

11.5 PROCESS

Although Chapter 13 will discuss how generator creation fits into the overall DSM definition process, it seems useful here to examine a few details and pass on some tips. You can view Chapter 13 as the strategy or game plan for the whole DSM solution team, while this section looks at tactics and instructions for the generator builder.

11.5.1 Creating and Testing Generators

As we mentioned at the start of this chapter, an important part of the process of creating a generator is knowing when to start. Unless you know the input and output required for a program, you are unlikely to be successful in building it. It is thus important to first wait until the modeling language is in an acceptable state. All the major concepts should be there and you should have made a few example models.

These example models will show that the modeling language works from the point of view of the modeler, which is the most important factor. You should also make sure when making the example models that the modeling language allows you to capture all the information you would need to build the application. In fact, it is sufficient to have enough information to be able to build an application corresponding to the example model: there are always choices involved when hand coding.

Next you need a matching pair of an example model and the corresponding working code. We looked at this in more detail in Section 11.1: either model an existing application or write the code for an example model. If you have the existing framework code, you will of course use that; otherwise, do not worry too much if you find repeating sections of code. Breaking those out into a domain framework is the subject of the next chapter; for now it is fine to do a little light refactoring as you go, but also acceptable to have larger pieces of code as boilerplate in the generators.

When you have the working code, use that as a basis for the generator. Actually, the working code itself is your first generator: simply make the generator consist of one

### PROCESS

large literal string! While that generator will produce exactly the right output for this model, we want to progressively break down that generator into smaller elements, with conditionals, loops, and reading values from the model. Follow the four steps of Section 11.1 to gradually reduce the generator down to its minimal form. You can run the generator and diff the output with the original handwritten code to check your progress.

Remember to keep the generator as simple as possible: push decisions and complexity down into the generated code (and later into the domain framework). If the generator needs more information, identify the extra problem domain knowledge that needs to be captured by the modeling language and find a minimal natural representation for it.

Once you have a working generator for that example model, take a snapshot of the model, generator, and code. Then try changing things in the model, and make sure the code still works but reflects the change. You can test both by running the code and by comparing the code with the snapshot. Start with simple changes like names that will be visible in the running application, and progress to more complex ones like adding an extra object. When you are satisfied that your generator works on the sample model, even when that is lightly modified, you are ready for the next step.

Building one application was never the plan, so go on and build a second, but keep the first application intact. Make the model for the second application similar to the first, varying by about the same amount as your boldest modifications did. Make sure that you can generate both applications and that both work. This may also be a good time to think about the modeling language and how you are going to enable reuse between models—a subject we cover in Chapter 15.

Working like this, you will soon have a fully functional generator. As you hit problems, use your ingenuity to find a way to solve them, remembering that there are several areas you can change: the models, the modeling language, the generator, the generated code, or the domain framework. One of these will almost always offer a good solution—if you are spoiled for choice, the later areas such as the domain framework are often the better places.

Later, when developers are using your modeling language and you want to enhance it or the generators, the same tactics still apply. You first need to know what information you want to capture and what is a sensible format for the modeler to input that and use it in models. Only after that do you look at what the code might be for an example usage, and work back from that code to the necessary generator change. Changes to the generator—whether refactorings or additions—can always be checked by comparing the output code from a known good model to its previous known good output.

You will probably find it useful to keep a set of example models and output for this purpose. If possible, do not let your work be based solely on these, but use real models built by the modelers. One good approach in the long run is to always use your example models to test changes, but each time use one other real model. Use a different real model each time: although you will spend extra time because that model will be unfamiliar, it forms a valuable part of the feedback loop between the language developer and users.

11.5.2 Sharing and Maintaining Generators

Generators are coupled with the modeling language on one side, and the domain framework, code architecture, and programming language on the other. A good principle of modularization is to keep coupled elements close together, particularly where a change in one often implies a corresponding change in the other. At the stage when you are first writing the generator, the first coupling is stronger: the modeling language is still likely to change, but for a given generator the programming language is unlikely to change.

The generator and the modeling language thus form a clear coupled pair, and it is good to keep them together. This guarantees that when a user receives an updated version of the modeling language, the latest version of the generator accompanies it— and vice versa, although that direction is less common.

While sometimes one may be updated without the other changing, there is no sense in allowing the generator and modeling language to lead separate lives. The vast majority of random combinations of generator versions with modeling language versions will simply fail to work. Separating them would be like separating a lex scanner from its yacc parser. While there is some degree of freedom, they are far from orthogonal.

The coupling of the generator with the domain framework is also notable, and once again it is mostly the generator that must change in response to changes in the framework. In this case, however, there may be several different parallel versions of the domain framework. The same generator produces code that runs with all of these parallel framework versions, each of which provides the same services. The different frameworks are used to abstract different underlying platforms, libraries, or components, hiding the differences behind a consistent interface.

An overview of the directions of information and change propagation is shown in Fig. 11.2. One modeling language, here the Watch DSM language, may have several generators (e.g., for different languages): the black arrows indicate the flow of information from the models in that language to those generators. Changes to the modeling language must be propagated to the generators, shown here by the thicker lines: information and change propagation flow in the same direction.

The situation on the framework side is more complex. One generator may produce output that works with several frameworks, but changes are propagated from the frameworks to the generator. In fact, the framework changes that must be propagated are those that change the interface between the framework and the generated code.

Keeping generators together with the modeling language—in the same tools and same files—ensures that when a generator is applied to a model, their versions are in sync and they share a consistent view of things. Building generators in the same tool as the modeling language also allows the generator editor to offer context-sensitive help and syntax checking on the many references to modeling language concepts. A tool could even update generators automatically if the name of a modeling language concept is changed.

Awell-architected tool can thus ensure version synchronization between modeling language and generators. Any serious tool should ensure version synchronization

### PROCESS

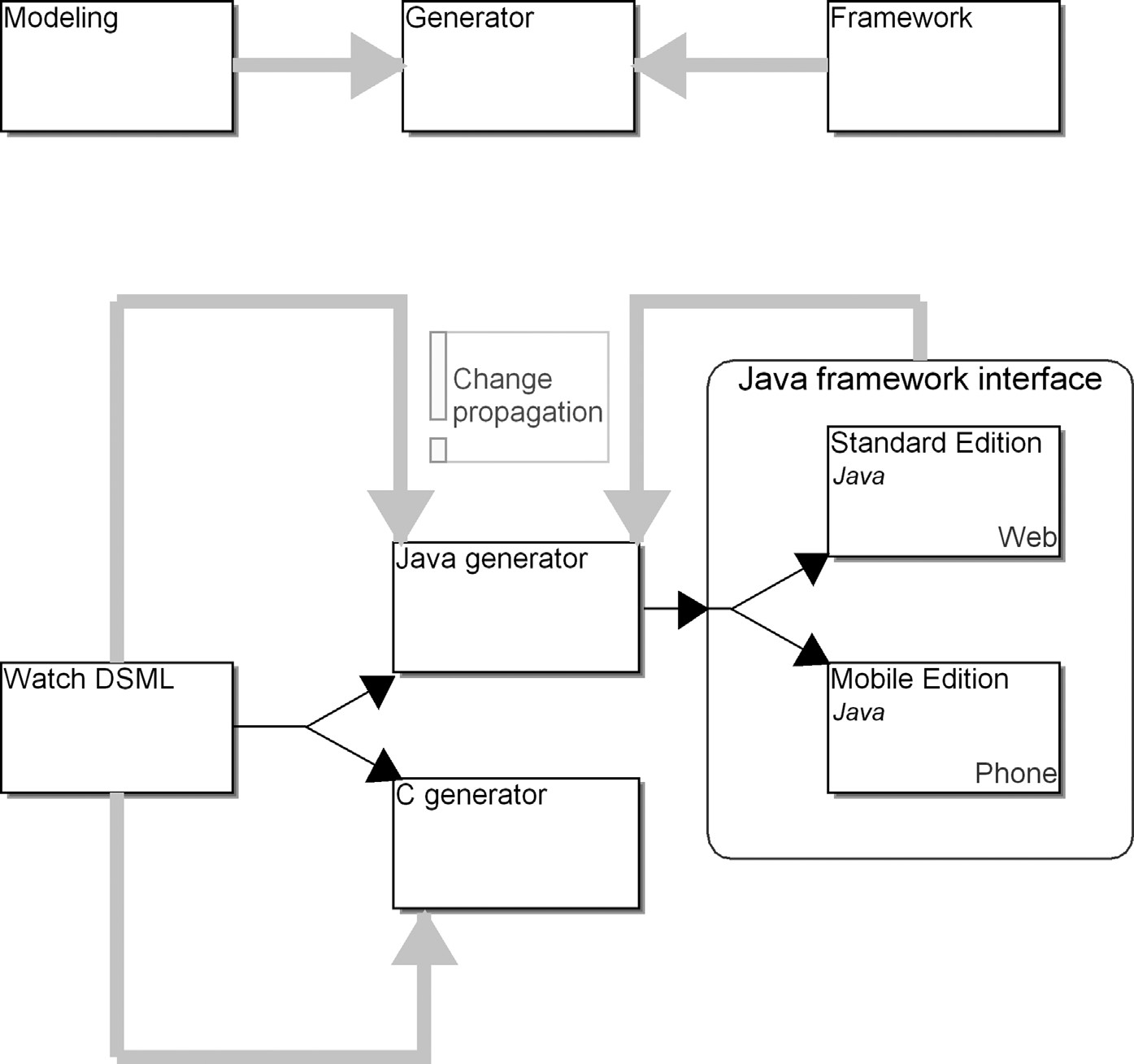


FIGURE 11.2 Coupling between modeling language, generators, and frameworks

between models and modeling language. This leaves version synchronization between code files, which is somewhat harder as these are normally pushed out away from the safety of the tool. A good idea is to make the generator output into the file the version information from all three sources: generator version, modeling language version, and model version.

Wherever possible, all the generated files needed to build an application should be produced at once, guaranteeing their version consistency. This is one of the additional benefits of a fully automated autobuild. Where that is not possible, one solution is simply to rely on the developers to do the right thing when linking together output from several generation runs. This, after all, is what has been the practice when hand coding—or indeed the situation was far worse, as different developers wrote different kinds of code for the same task based on a whole variety of personal and historical reasons.

The automated inclusion of version information by the generators does however also make it possible to reliably compare versions and ensure whatever level of correctness seems desirable.

11.5.3 Version Control of Generated Code

Do you version your .o files? No? Fine, let’s move on!

As mentioned above, with automatic generation and build you should not need to version the generated files. They are an intermediate product and can always be reconstructed from the primary source, the models. If, however, your DSM tool does not ensure correctness or combined versioning of models with their modeling language and generators, there could still be some scope for doubt. In other words, a poor DSM tool may not produce the same output twice from the same model: the modeling language or generators may have changed, even though you took the same version of the model from version control. In that situation it is important when versioning the models to include version information of the modeling language and generators.

In some rare cases, it may also be useful to version the generated code. For instance, if generation from the DSM tool is time consuming, and there are centralized nightly builds from hundreds of developers’ models, caching the generated code can speed things up. Another case could be where one set of developers builds the models and generates the output, and a second set takes that output and works with it—not changing it, but perhaps writing other code files by hand to work with it. It would generally be better to let this second group have access to the models, but this is not a perfect world.

The need for version control of generated code is thus the exception rather than the rule. In real world cases, our experience has been that the improved quality and consistency of generated code largely removes the need to worry about versioning— throughout the DSM solution. To a generation who have grown up believing in the safety net of version control, this may seem hard to believe at first. The fact is that current version control systems, practices, and requirements have grown out of the needs of hand-coded programs, and DSM removes many of those problems. Some parts of a DSM solution will still need to be versioned, but generally not the generated code.

11.6 SUMMARY

The generator forms the keystone of a DSM solution: it must interface with the modeling language, models, domain framework, and language to be generated. In some ways, building the generator is the most challenging task in DSM, since it requires a strong understanding of all of these areas and also of the generator facilities offered by the DSM tool. To ameliorate these potential difficulties, it is important to remember two things:

. Keep the generator simple: push complexity down into the domain framework.

. Do not try to build the generator too soon: wait until you have the modeling language, an example model, and the correct output to be generated from that model.

### SUMMARY

Making a proper code generator is definitely worth the effort. A sizable proportion of the productivity benefits of DSM are due to the generator, and in particular its ability to turn the declaratively specified solution from the models into the procedural form required by most of today’s programming languages. While in theory, and sometimes in practice, it is possible to interpret the declarative form at runtime, performance constraints tend to rule this out. This is no surprise: when we moved from machine code to Assembler, or Assembler to 3GLs, the majority of languages chose to transform the newer form to the older once at compile time, rather than continually at runtime.

The different types of generator facilities have their strengths and weaknesses. The sweet spot at the moment seems to be crawlers, which offer more power than model visitors or templates, while keeping the generator on a higher level of abstraction than the model API approach. More advanced template languages are also a good choice, especially where they are integrated in the DSM tool. Solutions that involve a separate generator program will always incur the additional cost of outputting the model in a format the generator can read, and parsing that format to rebuild the model structures on which the generator will work.

Generation is primarily thought of as focusing on code, but may also include codelike or declarative formats such as SQL or XML, documentation files, and simple text files. It can also be used as the basis for model checking reports, although the modeling language may be a better place for rules that should always hold, and for certain specialized analysis tasks it may be better to use existing external programs.

When generating code, the kind of code you want is determined by what you would write by hand, with perhaps some concessions to ease of generation. While this would appear to leave little in the way of generic patterns for code generation, experience shows that almost all code can be seen as either a flow model or a state model. While simple flow models can be generated with a naı¨ve recursive approach, anything more complicated tends to use functions, called either dynamically or statically, or then switch–case statements working similarly to functions.

Having the generator not only create the output files but also compile them and run them is a key ingredient in raising the abstraction level. Such an autobuild approach allows the modeler to go straight from models to the finished product, remaining at a high level of abstraction and thinking in the problem domain all the time.

Larger generators should be broken down as necessary into subgenerators for each modeling language, file type to be generated, and model element type. Repeated sections in generators should be refactored into subgenerators, just as with any program.

In many ways, the process of building generators is just a case of applying the generic parts of best programming practice. A generator takes a certain kind of input, normally an object structure, and processes it to produce a certain kind of output, normally one or many text streams. Only when the input and output formats are known, and an example of each is available, can generator construction sensibly begin.

The close coupling with the input format means the generators should be distributed and versioned with the modeling language. It is also good practice to have the generator annotate the output with version information about the models, modeling elements, and generator itself. While a good DSM tool will make most versioning issues invisible, having the information there may one day solve that mysterious “but it worked yesterday!” problem.