### CHAPTER 14

## TOOLS FOR DSM

In this chapter we will look at how to obtain tool support for your Domain-Specific Modeling (DSM) solution. By analogy with the evolution of other kinds of software, Section 14.1 will show the steps on the path from laborious hand coding of tools to simple configuration in integrated environments. Section 14.2 will take us on a whistle stop tour of the history of DSM environments, the current best evidence against the theory of evolution. In Section 14.3, which forms the body of this chapter, we will look in detail at the features that make or break a DSM environment. Section 14.4 will briefly look at the prominent tools available today.

14.1 DIFFERENT APPROACHES TO BUILDING TOOL SUPPORT

There are a number of ways you can build tool support for a new modeling language. By putting an ordering on them, we can consider them as levels as follows:

1. Write your own modeling tool from scratch,
2. Write your own modeling tool based on frameworks,
3. Metamodel, generate a modeling tool skeleton over a framework, add code,
4. Metamodel, generate the full modeling tool over a framework, (5) Metamodel, output configuration data for a generic modeling tool,

(6) Integrated metamodeling and modeling environment.

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

357

This is indeed a common pattern for the evolution of many kinds of systems. If we take an analogy with the output being a typeset document rather than a modeling tool, we can look at the history of word processing. While the match is not perfect, we can equate the modeling language to the formatting styles or template, the model to the document, the modeling tool to the editing software, and the generator to the printing software.

For word processing, levels 1 and 2 are lost in the mists of the past: it is a long time since anybody had to hand code their own editor or printer driver. PostScript, TeX, and LaTeX roughly correspond to the scale from levels 3 to 5. Only at level 6 with tools like Microsoft Word did it become generally possible to make on-the-fly changes to the styles, and have existing “instances” of those styles update instantly. Nowadays, virtually everyone is at level 6, with a few diehards proving their skills on level 5 or 4. Organizations have recognized the benefits and savings and created in-house templates for employees to use. Word processing is thus a mature technology, and mature tools have also been widely adopted.

We could also make an analogy with web applications: initially everybody had to write their own, then frameworks became available. The frameworks developed and moved from generating code to reading configuration files at start-up. Only with the advent of systems such as Seaside or Dabble, and to a lesser extent Zope or SharePoint, has it been commonly possible to create and test a web application in the same browser window. In this domain there may be good tools, perhaps even some that are both mature and good, but adoption has not yet progressed so far: many are still working with technology from the lower levels.

Regardless of the field, we would expect later levels to be better than earlier ones. Poor execution can however mar this: there are good and bad frameworks, and both kinds can be used well or poorly. All things being equal, a user who simply wants to get a job done will generally be best served by the more mature levels. Those who like to play with new technology for its own sake will be happy on all levels; those with a strong adherence to a particular vendor, language, or environment will take whatever is on offer there; those with a need for control will take the highest level where they still feel they have that control if they need it. Aside from personal predilection and “not invented here” prejudices, there can also be valid reasons for using a lower-level solution even when good higher-level solutions are available.

A key ingredient here is experience: for those creating their first word processing documents or stylesheets, Word is a better choice than TeX or PostScript. In many areas where the user finds Word does not support what he would like to try, the reason is more likely to be that it was a poor idea than that Word is somehow lacking. For instance, the user may find that the letter “i” looks too thin on his screen, and want to make it wider everywhere. This would be possible in PostScript, but Word offers no specific support for such an operation. The reasons are clear: fonts are generally designed so that character sizes are well balanced, and also any apparent imbalance is more likely just an artifact of the screen resolution. By not providing global percharacter settings, Word embodies this piece of experience; the apparent freedom of PostScript would be a bad thing for most users. However, an experienced typographer may well want to experiment with different kernings, and then the freedom of PostScript or TeX may be needed—at least until she finds an environment specifically built for typographers.

14.2 A BRIEF HISTORY OF TOOLS

Those who cannot remember the past are condemned to repeat it.

- George Santayana,

The Life of Reason (1905)

The history of tools for implementing support for new modeling languages has not been a pretty one. Unlike other software fields, such as operating system UIs, web browsers, compression or encryption, there has been little building on previous work. There seems to be no rational explanation for this; perhaps the main factor is to be found not from the field or tools themselves, but the mindset of the people who make them. Anyone who tries to define how others should build systems must be something of a control freak. Anyone who tries to define how such control freaks should define their modeling languages must be even more, ah, sure of themselves.

Looking back over the various tools clearly shows that later tools are by no means better than their earlier counterparts. Although there have been slight differences in emphasis, all have been attempting to solve the same problem. It thus seems useful to offer a potted history of the tools, their emphases, and how they fared. Particularly interesting in the context of DSM is whether the tools could be used by someone other than their creators.

14.2.1 1970s and 1980s

In the 1970s and 1980s, the search for a solution to the “software crisis” (Brooks, 1982) was focused in three directions:

. analysis of the whole process and concepts involved in building information systems (e.g., Chen, 1976; Brooks, 1982),

. application of more rigor in the early stages of projects by following explicit methods—modeling languages and associated processes (e.g., Gane and Sarson, 1979),

. documentation of the development of systems, often using computers—an idea going back as far as the early 1970s (Bubenko et al., 1971).

The branch of methods grew with astonishing rapidity, largely subsuming the other two branches, and producing huge numbers of methods of increasing complexity. Aside from leaving practitioners stranded in a “methodology jungle” (Avison and Fitzgerald, 1988), and for a long while forcing academics to limit their research to classifying the methods (Olle et al., 1982), rather than examining how they performed in practice, the growth of methods easily outstripped that of the computer tools that were built to help implement them.

The very early, text-based Computer-Aided Software Engineering (CASE) tools such as PDL (Caine and Gordon, 1975), PSL/PSA (Teichroew and Hershey, 1977), SEM (Teichroew et al., 1980), and SREM (Alford, 1977) had allowed changes to the modeling language supported, which gave users some possibility to maintain tool support in the face of rapid language evolution. However, newer languages and tools had adopted graphical representations and interfaces. While these were substantially easier to use, they were more complicated to specify, and thus CASE tools were no longer able to provide the user with facilities for changing the modeling language they supported. The CASE tools, heavily outnumbered by modeling languages, were thus forcing the users to adopt their built-in modeling language and process, rather than supporting the languages and processes from which the organization was already starting to see benefits. Conversely, the organization could continue with its own modeling language, substantially weakened by the lack of computer support, or even build its own CASE tool, a heavy investment in a venture in which it had no experience, and could often only make financially feasible by attempting to sell the resulting tool to others.

These conflicts, of course, did nothing to improve the image or practical benefit of the expected CASE revolution (Yourdon, 1986). CASE had been unrealistically trumpeted to be the “silver bullet” that would solve all software development problems. All too often, the response to the failure of CASE to provide such a solution was to blame the modeling language or the tool. This of course led to the development of yet more languages and tools: hardly likely to improve the situation.

14.2.2 1990s Overview

As we have observed, building a whole modeling tool from scratch is a large and complicated project, significantly too slow to keep pace with modeling language development. The solution to this conundrum thus lies in a tool that can be customized to support any modeling language. Two possible approaches to this were perceivable in 1990: effectively our levels 2 and 5 above. A modeling tool could be designed and built modularly, so that the minimum coding effort was required to change the part concerned with a particular modeling language. Alternatively, a tool could have the modeling language itself as data, rather than as code, and functionality could be provided for altering this data, in the same way as was done in the early text-based tools. A weakened form of the latter was a hybrid of the two: the modeling language was expressed in data or a mixture of data-like and code-like parts, which were transformed into code and compiled and linked with the generic modules.

The former solution was the one largely adopted by industry, in turning out new versions of an existing code base for new modeling languages. The approach however had flaws from the users’ point of view: only the vendor could make the changes, and the cost of such changes was high. While the reduction in work to make a modeling tool for a new modeling language was significant (one manufacturer claimed reuse as high as 90% (Rust, 1994)), the rate of such adaptation still proved insufficient to satisfy users’ needs. Furthermore, the cycle from requesting a change to using the modified tool was painfully long, and the customer was left highly dependent on the vendor.

The latter solution, called CASE shells (Bubenko, 1988) or metaCASE tools (Alderson, 1991), produced promising research prototypes and a few somewhat limited commercial products. These early tools were not widely taken into use, although the use of Systematica’s Virtual Software Factory metaCASE tool (Pocock, 1991) by IBM in building its BSDM support tool (Haine, 1992), and again by Heym and O¨ sterle in the construction of their MEET method engineering environment (Heym and O¨ sterle, 1993), provided practical proof that such tools could be useful. With some success in the Francophone world, GraphTalk (Jeulin, 2005) offered good pointers to future directions, although initially it only ran on special LISP machine hardware. Comparisons of these metaCASE tools (e.g., Marttiin et al., 1993; Goldkuhl and Cronholm, 1993) revealed that the process of metamodeling (Teichroew and Hershey, 1977; Brinkkemper, 1990)—configuring the tools to support a new modeling language—could be improved, as could the accuracy and breadth of the support for the modeling language in the configured tool.

MetaCASE tools of the 1990s tended to separate the metamodeling part from the modeling part. The description of the modeling language was written in a textual language, possibly compiled to some other form, and then fed into the configurable modeling tool, which then supported the new modeling language. Some more advanced tools provided partial graphical support for the metamodeling process, for example, MetaEdit and ToolBuilder. In tools of this type the graphical metamodel was made with the same modeling tool functionality as the resulting tool, but using a special graphical metamodeling language (whose metamodel had been bootstrapped from a textual description, and often had special links to tool functionality reserved only for that language). The resulting metamodel was first transformed into a textual language, which was then compiled as above.

This separation of metamodeling and modeling had an important drawback: it was not possible to interactively test the results of metamodeling immediately, because of the long transform–compile–link–run cycle to move from metamodeling to modeling. This was a significant hindrance to the metamodeling process, as if a problem was spotted while testing a modeling tool, the user had to exit the tool, restart the metaCASE tool, read in the metamodel, edit it, transform, compile, and link it to form the resulting modeling tool. These steps often required separate commands, manual text editing and external tools. The resulting modeling tool could then be restarted, and the existing models generally needed to be explicitly updated to use the new metamodel before the change could be tested.

Below we shall take a look at three tools of the 1990s. Others of course existed, and some tools born in the 1990s are still going strong (we will see current tools later in the chapter). The tools below were selected on the basis of their use in real-world industrial projects, their coverage of both metamodeling and modeling, and the availability of reliable evidence.

14.2.3 DOME (Honeywell)

The Domain Modeling Environment (DOME) was created by Honeywell Labs for use in its own research and implementation projects. It began as a project to build a Petri-Net editor but by 1992, after a few similar editors, it was apparent that it should be possible to generate such editors from a tool specification. This generation approach was called MetaDOME. The subsequent development of ProtoDOME and CyberDOME moved away from generation to interpreting the tool specification directly. This ability to store and edit metamodels directly as data meant that updates to the metamodels could be made while models were open.

DOME used a graphical metamodeling language called DSTL, which covered the main concepts and also some constraints and graphical behavior. The main concepts were Graph, Node, Port and Connection, but there were also specialized concepts related to Components and Interfaces, plus a special type, Archetype, for reusable objects. Extra behavior could be added in the LISP-like Alter language. Alter could also be used for transforming models, for example into code, but there was also a separate MetaScribe component for simpler generation. MetaScribe used a specification written in Word, apparently as a template-based generator, plus a separate output formatter to transform the output to a specific document format such as plain text or FrameMaker. DOME is described further in a number of articles (e.g., Engstrom and Krueger, 2000).

DOME was later released as open source, although the Smalltalk code has been static since 2000. Relying only on older Smalltalk versions meant the user interface was rather Spartan and nonstandard. The use of files for models was seen as a limiting factor for the adoption of DOME by larger teams: only a single user at a time could edit a file, and there was poor support for breaking a model into separately editable chunks. To address some of these limitations, a rearchitecture and reimplementation in Java has been underway since 2003.

14.2.4 MetaEdit (Jyva¨skyla¨ University/MetaCase)

MetaEdit (Smolander et al., 1991) was the predecessor of MetaEdit+. It consisted of a generic modeling tool whose modeling language support was provided in binary metamodel files. Its metamodeling language was based on OPRR (Smolander, 1991), exhibiting a rare instance of reuse of existing work: OPRR was originally made by Welke (Welke, 1988), and used in the QuickSpec tool (Meta Systems, 1989).

While metamodels could be built purely textually, MetaEdit included a graphical modeling language containing the OPRR concepts, with which users could graphically define their own modeling languages. The symbols for the new modeling language were defined in a separate graphical symbol editor. The editors could generate these two parts of the metamodel in the textual metamodeling language, and the user would join these parts by hand to form one file. This file could then be run through the Moffer metamodel compiler to generate the binary metamodel file that drove the generic modeling tool part. If a metamodel was changed, existing models based on it could be explicitly updated to use the new metamodel: the actual update was performed automatically. The ease and incremental nature of metamodeling were recognized in the tool comparisons as the main advantages of MetaEdit.

MetaEdit had several limitations, mostly in its modeling tool functionality. Each model file was limited to a few tens of diagrams, all of the same type, and there was no linking between model files. This prevented proper implementation of most modeling languages containing several integrated diagram types. An odd implementation of relationship metamodeling required creating several near-duplicates of relationship types in some not infrequent cases (Kelly, 1995), also complicating modeling use with extra types. The conceptual–representational distinction was rather weak: each model file was conceptually one large graph, of which subsets were visible in different diagrams. This ran contrary to the user perception of a model as consisting of several graphs, each represented in one diagram. There was no support for complex properties, effectively ruling out true modeling of the new object-oriented methods with their Classes containing collections of Attributes, each with its own properties. Complex object support was limited to a simple explosion construct, representing a free-form link between an object and another diagram in the same model.

Despite these limitations, MetaEdit proved somewhat successful as a metaCASE and modeling tool, with users numbering in the thousands. It was also used as a tool in other research projects, for example, Jarzabek and Ling (1996) found it useful in developing a BPR modeling language and its tool support. The ease of metamodeling seems to have been a major factor enabling others to use MetaEdit without intervention or hand holding from its creators.

14.2.5 TBK/ToolBuilder (Sunderland University/IPSYS/Lincoln)

The TBK (Tool Builders Kit) framework and ToolBuilder metaCASE system was originally reported in Alderson (1991) and was later commercialized by IPSYS. The system consisted of three components:

. The specification component—used to create the specification of the tool;

. The generation component—used to transform the specification into parameters for the generic tool;

. The run-time component—the generic modeling tool itself.

The metamodeling language was ER extended with some constraints and the ability to have attributes whose values were derived from other attributes. It allowed triggers on events applying to attributes and relationships. TBK definitions were in the form of textual files in four languages:

. Data definition language (DDL), which specified the basic objects and the attributes they had,

. Graph description language (GDL), which specified the graphical symbols, but also simple relationship and decomposition rules,

. Format description language (FDL), a low-level description of the individual windows, buttons, menus, and so on that make up the user interface; . Text layout language (LL), describing a structured text format of user-enterable text fields interspersed with fixed text and punctuation.

DDL and GDL appear not dissimilar to the corresponding parts in MetaEdit’s textual metamodels, although with a rather more involved syntax. The LL text layout language appears a useful addition, although little is mentioned about it in the documentation and articles. The need to specify tool behavior in FDL, however, seems to be a symptom of the level of abstraction being closer to that of a framework than a true metaCASE tool: it should be possible to provide a good generic interface, or generate one based solely on the DDL and GDL input.

Toolbuilder added a graphical front end, the METHS system, that could generate parts of the information captured by these textual languages. More detailed editing was performed in the textual languages, in the EASEL language (a rather simple 3GL), and through user functions written in C. The results were partially compiled and partially interpreted by DEASEL, the generic modeling tool runtime. DEASEL provided standard modeling tool functionality and supported multiple users on a true repository.

Thus ToolBuilder appears to have provided a usable metaCASE system, but the time required to build support for a modeling language was long, even according to IPSYS’s own marketing material:

Even a completely new method can take only man-months to implement, with prototypes taking man-weeks. The rapid prototyping nature of ToolBuilder means that demonstrations can be created in man-days.

This slowness was probably because of the complicated textual languages, and the separation of the modeling and metamodeling tools. Also, while it was of course a benefit that DEASEL provided basic default modeling tool behavior, the description gave the impression that these defaults were probably insufficient for most actual modeling languages, requiring coding in EASEL or C to specify tool operations.

While ToolBuilder did not succeed as a metaCASE tool, two modeling tools based on the work on ToolBuilder had more success. A precursor of TBK was used to build the HOOD toolset used on the Eurofighter project, and ToolBuilder was also used to build the Kennedy-Carter I-OOA Shlaer-Mellor toolset. The latter took 12 months to build; the number of developers involved is not revealed.

According to one of the main figures behind ToolBuilder (Alderson, 1997), a major factor leading to IPSYS’s eventual business failure was a lack of in-house expertise in the popular modeling languages of the time. This led to a series of projects where IPSYS would give a customer one of its developers in a project to create support for a new modeling language. The customer would normally receive free use of the resulting tool, and IPSYS would try to sell it to others. Sadly, in no case did this work as planned: either customers were not sufficiently invested in the modeling language, or IPSYS lacked the experience necessary to market and sell a tool for what was to them an unfamiliar modeling language. The remains of IPSYS were bought by Lincoln Software in 1995, whowere in turn bought by PeerLogic in 1999, who themselves were bought by Critical Path the next year. ToolBuilder disappeared along the way, although there was a project to include part of the ToolBuilder-built Engineer product into IBM’s WebSphere, to provide a web front end to CICS transaction servers.

Technically, the main problem with ToolBuilder seemed to be the large resource cost of building support for a new modeling language, coupled with the specialist knowledge required to use ToolBuilder. There seems to have been no case where customers were able to build anything other than academic prototype modeling language without experts from IPSYS working alongside them. As Isazadeh’s thesis puts it (1997):

At run-time most of the functionality of the tools is provided by C functions [handwritten by the metamodeler]. ... In general, a conclusion in working with ToolBuilder by a long-time user [Sunderland University’s Ian Ferguson (1993)] is the extreme complexity of extending the functionality of the tools beyond the default operations which, together with poor documentation, requires a lengthy learning curve.

14.3 WHAT IS NEEDED IN A DSM ENVIRONMENT

When looking at what is needed in a DSM environment, we can take some things for granted. First let us look at the minimum facilities for metamodelers:

. Specify the object and relationship types declaratively,

. Specify declaratively a list of properties for each object or relationship type, with support for at least string and Boolean property data types,

. Specify basic rules for how objects can be connected by relationships,

. Specify symbols for types, whether graphically, declaratively, or in code,

. Ability for a generator to access the models,

. From these specifications, create a basic modeling tool.

The modeling tool thus created must offer modelers at least the following facilities:

. Store and retrieve a model from disk,

. Create new instances in models by choosing a type and filling in properties,

. Link objects via relationships,

. Lay out the objects and relationships, either by dragging or automatic layout,

. Edit properties of existing objects and relationships,

. Delete objects and relationships

The history of tool development contains numerous instances of small research projects creating this kind of prototype DSM environment (or metaCASE tool, in 1990s terms). Developing such a tool to the stage where you can produce screenshots for an article is relatively simple: 1–3 man-years is enough, depending on implementation technology and previous experience.

With the recent realization of the benefits of model-driven development, and the need for its modeling languages to be domain-specific, we are currently seeing a new explosion of such tools. Most of these are again from research projects, or their modern-day distributed cousin, open source projects, but a few are also commercial projects. The year 2006 saw the release of the largest number of new tools like this, even exceeding the metaCASE boom of the 1990s.

These prototypes and “version 1.0” tools generally support a single user, one modeling language at a type, simple metamodels focusing on objects with basic properties and relationships, and symbols with a single graphical element and a label. Generator facilities will be based on text-to-text transformations or handwriting code to read models. The resulting modeling tool will normally be missing the majority of functions that users expect from a graphical editor.

Compared to creating the initial prototype, turning it into something that could be used in the real world is a much harder task. Building such a system without significant personal experience of industrial scale DSM appears to be a near impossibility. This is sad, but not surprising: if building a metamodel requires the top expert developer in a domain, building a meta-metamodel for all such top developers requires the very highest levels of experience as well as intelligence and skill. The only other hope is dumb luck, but then the current authors probably used most of that up already, leaving little for newcomers!

We can thus assume the basics: a tool maturity level of at least 3, and the basic metamodeling and modeling facilities mentioned above. In the rest of this section we will concentrate on what is needed after such a generic “version 1.0.” Since each tool has its own focus, even version 1.0 will contain some of these features, and possibly even several from a given category. There is no stigma to be associated with the term “version 1.0”: every tool will have its first version. Similarly, there is not necessarily a benefit to being at “version 7.5”: in many ways, the thought and understanding already discernable from the first couple of versions are a better indicator of success than a high version number.

Clearly, the authors are associated with one particular tool, which happens to be one that has been developed for longer than many others—nearly 20 years at the time of writing. Unsurprisingly, given that time frame, many of the things we identify as useful have already been implemented in the tool. Indeed, the majority of these can already be found from the original articles on MetaEdit and MetaEdit+ (Smolander et al., 1991; Kelly et al., 1996): this is thus not a cheap attempt to list all the features of the current MetaEdit+ 4.5. Conversely, some things we originally thought useful and implemented have turned out not to be so valuable, adding little or even being detrimental. Although those features may be found in earlier publications, and some borderline cases may even still be present in the tools for the sake of existing customers, they are thus omitted from this list. Finally, although the list will necessarily be most colored by our own experiences and subjective viewpoint, we have always encouraged a frank exchange of views with other tool developers, and hopefully have been able to learn from their experiences and solutions—whether or not we have implemented similar features in MetaEdit+.

We will divide the features into the following categories:

. Meta-metamodel

. Notation

. Generators

. Supporting the metamodeler

. Generic modeling tool functionality

. Tool integration

The first three of these are the most important, as they form the foundation on which the remaining three are built. For the first two items, Chapter 10 looked at how to create a modeling language and notation, but not so much at the details of what features tools should offer for these. Chapter 11 however already discussed the various kinds of generator facilities found in tools (Section 11.2), so here we shall focus mostly on the first two items: the meta-metamodel and notation facilities.

14.3.1 Meta-Metamodel

The meta-metamodel is in effect the metamodeling language: the set of concepts provided to the metamodeler for building his metamodel. Just as each element in a model is an instance of a concept in the metamodel, so also is each concept in the metamodel an instance of a concept in the meta-metamodel. If the reader can cope with a shift in metalevel, we can say that the meta-metamodel is in fact a domainspecific modeling language for describing modeling languages. As such, it should aim tohaveagoodsetofconceptsforcoveringallthecommonlyfoundartifactsofmodeling languages. It will probably also have to be built specifically for this task: an existing general-purpose modeling language will tend to be at too low a level of abstraction.

For a first attempt, many people will still try to use an existing general-purpose modeling language: ER in the 1980s and early 1990s, or UML (or its MOF subset) nowadays. We can assume something like this as offering the basics of a metametamodel, which we shall not cover further here. Table 14.1 shows these basic concepts from ER—designed for describing databases, UML—designed for describing object-oriented code, and OPRR—designed for describing modeling languages.

Below we shall look at the extra features whose support will make or break a metametamodel. “Make or break” may sound strong, but of all the categories this is the one that has the greatest effect. If a meta-metamodel is lacking in some area, you will have to twist your modeling language to fit it. This will obviously be a burden to you when metamodeling, probably when building generators, and certainly for the modelers. Worse, even if the tool is later updated to include the feature you would have needed, there will be little chance of making the wholesale changes necessary to take

TABLE 14.1 Basic Meta-Metamodel Concepts

|  |  |  |  |
| --- | --- | --- | --- |
|  | ER | UML | OPRR |
| The basic model elements: | Entity | Class | Object |
| ... are connected by these: | Relationship | Association | Relationship |
| ... and both store information in these: | Attribute | Attribute | Property |

advantage of it. Updating existing models for such a large change in a metametamodel will often be rather like porting object-oriented code to a functional programming language.

Contrast this with improvements in the other categories, for example, notation or modeling tool functionality: there your modelers can benefit from the updates easily, without having to update models. It is also worth noting that in our experience as tool vendors, the meta-metamodel is something that is hard to change later: all the code of the tool is implicitly dependent on it.

ExplicitConceptofGraph Early versions of tools tend to handle the objects of a model as one large cluster. This is rather like writing a program in one code file: it will work fine for a small case, but will not scale to be useful in real-world cases. There must thus be an explicit concept of a graph: a set of objects and associated relationships. This allows a model to be divided into submodels, with obvious benefits for manageability and multiuser use. It also provides essential capabilities for reusing parts of models, for example, building product variants from a subset of all graphs.

Larger-scale concepts can also be added above the graph. Just as a related set of graphs forms a model, a related set of models can be collected into a project, and projects themselves can be collected into repositories. The exact details are not important, but something extra is often needed in addition to the basic concept of graph.

Objects as Property Values In a version 1.0 DSM environment, objects in models can often only have properties whose values are simple strings, numbers, or Booleans. Surprisingly, even the humble string can cover almost all the data storage needs of most modeling languages. Even numbers are relatively rare: they tend to be needed more in the metamodel constraints or in systems generated from models. In models, most fields that would commonly hold a number tend to benefit from being strings, as this allows the substitution of a variable or named constant rather than a literal number.

The most common type of property value after strings is a reference to another object. This provides a powerful way to integrate models: by referring directly to another object, rather than just typing its name, a modeler solves a major maintenance headache. If the name of the referred object should change, all other model elements using it will still refer to the correct object. In some cases, of course, the extra level of indirection provided by a string can be useful, but the facility to refer directly to an object when necessary is vital. Any tool supporting strings will allow references by name; only by also supporting direct references can the metamodeler be given the choice of which method is most appropriate in each case.

Constraints on Property Values Although most properties are strings, they are by no means all free-form strings. In early versions of tools, there is often noway to check the input for a given property, and generators tend to spit out the value as is. If the value contains characters that are illegal in the context of the output file, this will lead to errors either during compilation or, worse, at runtime. For instance, a string property to be used as a variable name may contain a space.

There are two possible solutions, discounting the idea that modelers should simply remember which characters to use in which fields. Either the generator facility must allow filtering or translation of strings from the model, or the metamodeling language must allow the specification of checks on property values. The former is probably better, as it allows more human-readable names in the model, but the latter approach is also useful. There are various ways to provide such support: MetaEdit+ uses regular expressions to specify legal values; IPSYS TBK had a text layout language that could handle simple cases, for example, free-form text separated by commas.

Constraints on property values can also look beyond the string itself to the wider context of the object and even graph where it is being entered. For instance, there may be a constraint that no two objects of a certain type could have the same name, either globally or at least not in the same graph. While such constraints could be handwritten in code, they occur frequently enough that supporting them in the metamodeling language is useful.

ExplicitConceptof Role Fewfirstversionsoftoolspaysufficientattentiontothe connections between objects. Even a casual examination of areas where graphical languages provide benefits over their textual counterparts reveals that visual links between objects are a major advantage. Anybody who has tried to read an XML file where elements need to refer to each other across the tree structure will be aware of how difficult it is to see such information: a graphical representation shows this information instantly. Since the graphical representation must also show the tree structure, therewill be more than one kind of link between objects. These links must be represented differently: different line thicknesses and colors, arrowheads, and various decorations.

The different kinds of links also provide an important source of the semantics and rules of the modeling language: the differences between them are more than skin deep. As the semantics often depend on the direction of a connection, there must be a distinction according to the role an object plays in the connection: is it a superclass or subclass, a container or contained element. If we restrict ourselves to binary relationships, it is possible to consider this information as part of the relationship. Each relationship will have its own information, and a set of information for each end: what object it connects to, and any properties describing that connection. If we look a little further and allow relationships connecting more than two objects, that set of information is seen more clearly as a concept it its own right: a role.

N-Ary Relationships In a binary relationship, it appears that the arrowheads are part of the definition of the role the object plays in the relationship, but the line itself is part of the relationship. Looking at a ternary relationship, a connection between three objects, we see that the line too is actually a feature of the role. The meeting point of the three lines in a ternary relationship is the relationship itself, which may have a graphical symbol. Each of the roles can also have its own properties, as can the relationship. Applying this back to the binary relationship, we can see that this more general case applies perfectly well there too: the line type may almost always be the same for both roles, but the symbols and properties will often be different for each role and the relationship itself.

The simplest kind of n-ary relationship is one where a role can be repeated multiple times, for example, multiple “To” roles in a Data Flow Diagram, or multiple “Specialization” roles to subclasses in a Class Diagram. This can be specified as the cardinality of the role in that kind of relationship. More complicated cases, for example, the Transition in the WatchApplication modeling language in Chapter 9, will have several different roles, each with their own cardinality, and each with a different set of object types it can connect to.

Weaker metamodeling languages that only support binary relationships generally attempt to make the excuse that an n-ary relationship can be replaced by a new object with a relationship to each of the connected objects. This is a poor workaround, as it loses the ability to check the legality of the relationship, or at best makes such checks much harder. Of course, most relationship types in modeling languages will be binary, as will most instances of them in models; still, having the choice to use n-ary relationships when you need them is a useful feature.

Reuse of Objects in Models Being able to reuse objects directly by reference is an important part of the productivity gains of DSM. The same object can be visible in many places, but in each case it really is one and the same object. An object can be reused in many different ways:

. As a property value of several objects, as seen above.

. By appearing several times in the same graph representation, as a notational convenience. In a diagram, this can be to prevent lines crossing or becoming too long, as in the Watch examples in Chapter 9. In a matrix, the same object will often be found on both axes, to allow relationships with itself to be shown. . By appearing once in each of several different representations of the same graph. This allows radically different views on the same graph, for example, as both a diagram and a matrix, or as one diagram emphasizing inheritance and a second diagram emphasizing aggregation.

. As an element of several different graphs of the same modeling language. This allows a similar kind of reuse to functions and components in textual programming languages. See, for example, the states representing WatchApplications in Chapter 9.

. As an element of several different graphs of different modeling languages. This allows method integration: building a large model with different viewpoints or aspects described in different modeling languages.

Reuse also changes some preconceptions that we may have from simpler cases of modeling. For instance, looking at the case where an object is reused in several graphs, it is clear that an object cannot itself store all of its connections to other objects. In one project an object A may connect to B and C, but when A is reused in a different project that may have changed. The information about connections should thus be stored in each graph, and when we ask questions like “what are the subclasses of X,” we must provide extra context: “... in graphs of project 1.” To take a concrete example from Chapter 9, we cannot simply ask “what application follows the Time application?”, since the Time application is reused in several Logical Watches. We must provide more context: “... in the TASTW Logical Watch.”

Links to Subgraphs A main reason to support multiple graphs is to allow the partitioning of a large model into several graphs. An immediate extension of this is to build a top-level graph where each object links to the corresponding lower-level graph. This kind of structure is found in many modeling languages, for example Data Flow Diagrams (Gane and Sarson, 1979) and the various kinds of hierarchical state transition diagrams.

The metamodeler must be able to specify which object types can link to subgraphs, andofwhichtypes.Insomecases,relationshipsandevenrolescanalsohavesubgraphs. There is also a wide variety of semantics associated with the subgraph links: Iivari (1992) finds five Boolean dimensions, for example whether the subgraph is owned exclusively by one object, or several objects can all have the same shared subgraph. The possible uses of subgraphs seem to vary along a scale of how formal the link is. At the stricter end of the scale an object may only have one subgraph, and that link is the same wherever the object is reused. At the freer end of the scale the link is more like a note or hyperlink: an object may link to several different graphs, and that set of graphs may be different when the object is reused elsewhere.

Explicit Concept of Port Modeling languages for describing hardware often mimic electronic circuit diagrams in allowing an object to have a set of ports that connections must attach to. These ports may have different semantics and rules, for example, a “power in” port on one object must be connected to a “power out” port on another object. A port is thus a feature of an object type in a metamodel, and for ease of use it is generally associated with a particular visible node on the perimeter of the object symbol. Ports lend themselves to rules that are expressed over the whole modeling language: “in” ports must always connect to “out” ports; a “5 V” port cannot connect to a “110 V” port.

Some software component modeling languages have adopted a similar convention for the services provided and required by various components. In contrast with the first kind of ports, which are defined as part of the modeling language, these component interface ports are added by the modeler to each object as he sees fit. Since the ports are then part of the model, not the metamodel, they cannot be constrained in the same way by rules. Similarly, all such ports have the same semantics, reducing the ability to generate useful code from them.

Modeling languages generally fall into two categories: those that primarily use role and relationship types to express rules, and those that use ports. Where ports are used, there is often no semantic need for different role types: distinguishing between “To” and “From” is not done by role type, but by “In” and “Out” ports. Ports and roles can still usefully be combined for notational convenience: a separate “To” role type to show an arrowhead when connecting to an “In” port, or constraining a “From” role type to always leave from the right or bottom edge of a symbol.

Language for Arbitrary Constraints The rules for connecting objects via relationships, roles and ports are a vital part of a modeling language. A good metamodeling language will allow the metamodeler to easily specify a range of common rules. Since these types of rules will be known by the generic modeling tool, they can also be checked efficiently.

No matter how wide the set of rule types offered, it will always be possible to invent a rule that cannot be expressed with them. It can therefore be useful if the tool offers the metamodeler a language to express arbitrary constraints. Using this language will however be harder work for the metamodeler, and checking the constraints will be harder for the tool—particularly in large models or multiuser situations.

Various kinds of language have been used for this task:

. Tool programming language

. Higher-level or logic language

. OCL

. Generator language

The naı¨ve initial choice is often the same programming language the tool itself is written in. While clearly easiest for the tool vendor, this language will not be well suited to the task: it knows nothing of the domain of modeling languages and models. Writing constraints will require a large amount of low-level code, and debugging this mass will be a major difficulty.

For this reason several tools have turned to higher-level programming languages such as LISP or Prolog. A problem here is that the metamodeler will normally have to learn this new language, and these languages are generally found to be hard to pick up. Perhaps the best such language for this task would be Smalltalk, since most objectoriented programmers can learn it quickly, and it allows domain-specific constructs to feel like a natural part of the language.

Several patterns from Smalltalk were indeed used as the basis for OCL from the OMG (2006). OCL includes Smalltalk-like collection operators for transforming and filtering collections, plus a limited set of basic string, arithmetic and collection operators. It also include some extensions useful for constraints, for example one():

collection->one(iterator|body\_expressions)

which returns a Boolean stating whether the body\_expressions evaluated to true for exactly one element of the collection.

While OCL was intended primarily for use in models to specify constraints on running systems, it can also be used one metalevel higher: in metamodels to specify constraints on models. As might be expected when a language is used for a somewhat different domain than intended, some tasks have to use a rather low-level approach, and in other areas the language appears rather bulky. Some tools such as GME or XMF-Mosaic have indeed found it necessary to invent their own “versions” of OCL to address these problems, raising the question of whether such a language can claim to be standard anymore. A significant problem with OCL is its tight coupling with UML and MOF: to use OCL, you must use MOF as your meta-metamodel; using MOF, you can only build modeling languages that share a number of similarities to UML.

A different approach is to use the same domain-specific language for constraints as for code generation. This allows the use of a language specifically designed for navigating over models, without forcing the user to learn yet another language. If the output of generators contains links back to the corresponding model elements, the warnings and error messages output by failed constraints can be clicked to go straight to the offending place in the model.

An interesting combination is to have the modeling language symbols directly contain elements for showing failed constraints. This places the error exactly where it should be, and makes it visible at the same time as it is caused—no need to explicitly run a constraints check. There are still cases where some constraints should be left to be run explicitly and as a test before generation: for example, if the constraint will often be broken, or checking it takes too long for execution on the fly.

14.3.2 Notation

If the meta-metamodel sets bounds on the abstract syntax of your modeling language, the notational features offered bound its concrete syntax. For tool vendors, the notation side definitely seems to be a harder task. Even the most basic part, the facilities for defining the graphical symbols, has proved troublesome to nearly all tool builders.

Representational Paradigms The clear majority of DSM languages have been graphical diagrams: nodes and edges, boxes and lines, bubbles and arcs. The same modeling language, or more exactly its abstract syntax, can however be represented in a number of different representational paradigms: as a matrix, table, or structured text.

A matrix representation shows the objects on the axes of the matrix, like row and column labels in a spreadsheet, and a relationship between a pair of objects is shown in the appropriate cell. The matrix is thus useful for focusing on relationships, particularly where there are binary relationships each connecting an object of one type to an object of another type. An example of this was seen in Figs. 9.5 and 9.6 for the Watch family diagram. In the former, each WatchModel object had properties referring to the Display object and LogicalWatch object from which it was composed. In the latter, these structure were turned into WatchModel relationships that directly connected Displays and LogicalWatches. Figure 14.1 shows a diagram and a matrix displaying the relationship approach, and a table displaying the property approach.

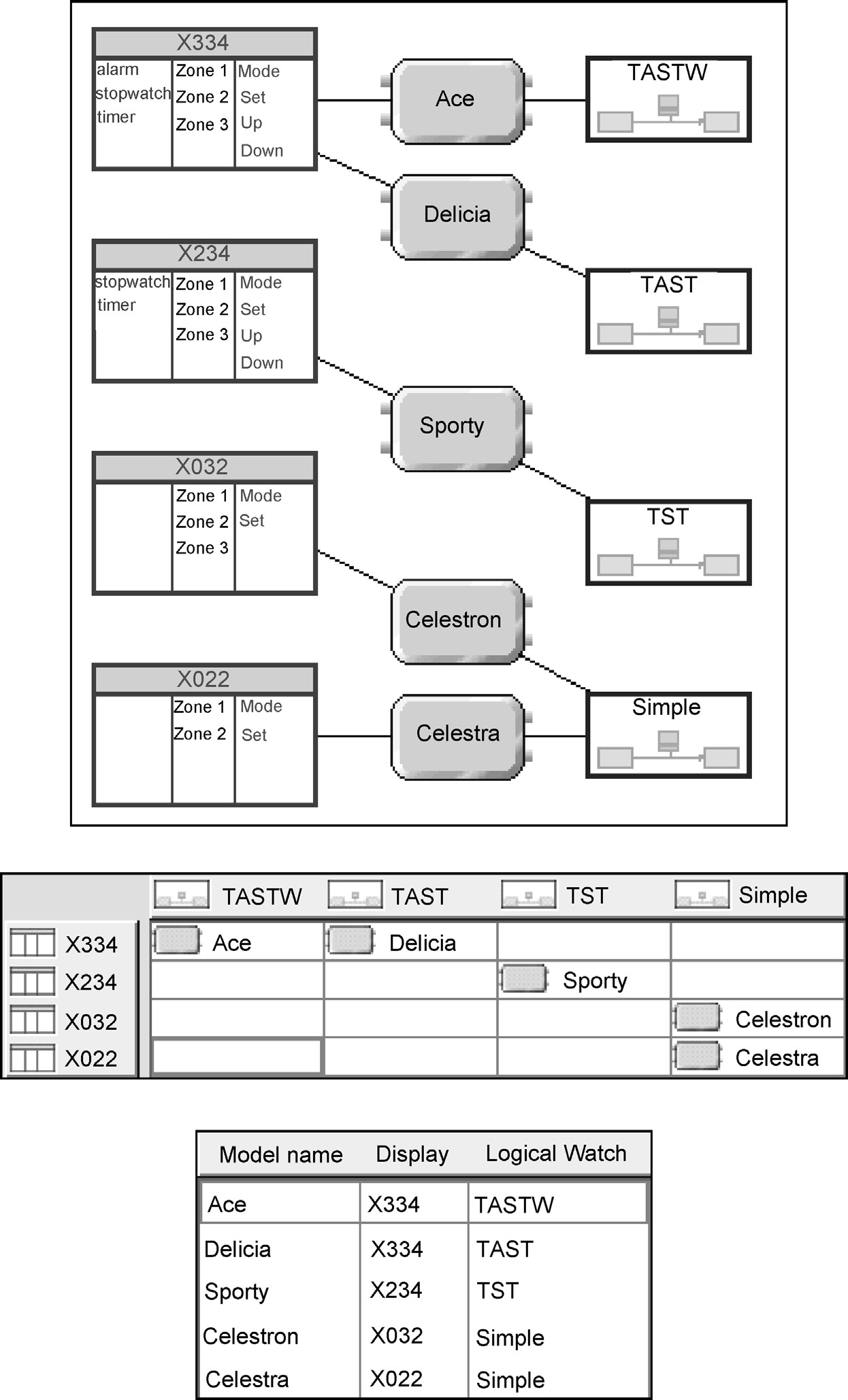


FIGURE 14.1 Diagram, Matrix, and Table representations

As should be clear from the picture, sometimes the table and matrix representations have significant advantages in terms of compactness and scalability. Mapping between four Displays and four LogicalWatches works reasonably in a diagram: in this case the lines do not even cross each other. Somewhere between four and eight this kind of diagram would probably break down, if the WatchModel connections were random. The earlier diagram, Fig. 9.5, would scale slightly better, but is perhaps less clear visually. A matrix would cope with up to 15–20 of each kind of object, and many more WatchModel relationships, while providing a good view of both objects and relationships. A table would scale to over 50 WatchModels, but offers less help in visualizing which Displays and LogicalWatches are used where.

Graphical Symbol Editor Given that the direct manipulation vector graphic editor has been around since 1961 (Sutherland, 1963)—8 years before ARPANETand IBM’s first 1 KB RAM chip—it is hardly impressive to say that of all the tools, to our knowledge only two have provided metamodelers with a graphical symbol editor: MetaEdit and MetaEdit+. This must surely change in the near future, but the failure of earlier tools leads us to suspect that it may take a little time before the other current tools get it right.

Getting it right is however worth the effort, for the sake of both the metamodeler and modeler. Two of the most common criticisms of tools without a symbol editor are the lack of freedom in defining symbols using the facilities offered, and the massive amount of time it takes to program the display of such symbols by hand. Even with a symbol editor, significantly more time is spent building the symbols than building the abstract syntax of the metamodel: concepts will always be simpler to handle than representations.

Without a symbol editor, the amount of time required to define symbols will often approach the ridiculous. For instance, Eclipse’s GEF comes with a simple example modeling language for logic circuits: AND, OR and so on. The code for the editor comes to over 10,000 lines, mostly for the graphical symbols. For the average Java programmer, that represents a little over one man-year of work. In MetaEdit+, implementing the same language took just one hour: over 2000 times faster. Even an author of the IBM Redbook on EMF and GEF (Moore et al., 2004), Anna Gerber, accepted the difference of roughly three orders of magnitude, although she would expect the actual figure to be 600–700. Mind you, she is probably faster than the average Java programmer!

Some tools have tried offering a limited range of preprogrammed symbols, configurable by size, color, and so on. While a similar approach has worked well for rules, with symbols its limitations are reached too quickly. The difference probably lies in the fact that the kinds of rules that are needed are to be found from the domain of metamodeling, with elements such as objects, relationships, and properties, whereas the kinds of symbols that are needed are to be found from each modeling language’s own problem domain. Trying to fix the set of possible symbols ahead of time means restricting them to simple geometrical figures or those used in other, mostly generic, modeling languages. An important part of DSM is that the symbols should be evocative of the domain concepts they represent, and to do that requires free combination of the basic graphical elements—lines, curves, text, and so on—to make domain-specific symbols.

There seems little point in describing the features of a graphical symbol editor here: it should simply behave like a normal vector graphics editor. The difference is in the integration with the modeling language definitions: a given symbol is defined for a particular object, relationship or role type. Text elements will thus normally be references to the properties defined in the type, and the editor should be integrated with the definitions to make selecting the right property easy.

The editor should also be integrated with the modeling tools, so when a symbol is altered, existing models update immediately. Seeing a symbol in several places in a model, possibly with different scales and zooms, gives a much better impression of how well it works than seeing it once in the symbol editor, often zoomed to make editing easier. This once again emphasizes the importance of having the metamodeling and modeling tools integrated in the same environment.

Role Symbols The symbols for roles are something of a special case, because they must rotate with the line over which they are displayed. Tools without a true concept of role will often offer just a few simple choices of arrowheads, but even a cursory look at existing modeling languages will reveal far more variation. Since vector graphics can be rotated with relative ease, it seems simplest to allow the same freedom to roles as to other symbols.

Text elements are somewhat problematic in role symbols. Most modeling languages have found that text works best when displayed horizontally. A slight angle or incline to text along a role line is no great problem, providing the font can be rendered suitably smoothly (still a challenge in many cases). Anything over 45of incline tends to render models much harder to read, with the user having to crick her neck back and forth at a different angle for each role. Since that represents half of the possible role line angles, rotating the actual text content tends to be a poor idea. With Manhattan (i.e., perpendicular) routing the situation is even worse: half of all role lines are at the absolute worst angle, vertical.

If the contents of text elements are to be displayed horizontally, there is the question of how their position and size react to changes in the role line angle. Most people’s initial attempt will have the text offset above a horizontal line, but this leads to problems when the line is rotated to be vertical. If the text box is also rotated, it will now have many rows each of one character. If the text is not rotated, a long text stretching out perpendicular to a vertical role line will lead quickly to confusion, with texts crossing several such role lines. Even with shorter texts, having the text offset above a horizontal line will mean that when rotated to point upward, the text will be offset to the left of the line.

Some of these problems can be seen in Fig. 14.2, which shows two different symbols for a role at the top, and then effects in a model below. The tool here is using what we have found to be the best approach to rotating text elements in role symbols: do not rotate the content or boundary of the text, but allow its midpoint to rotate with

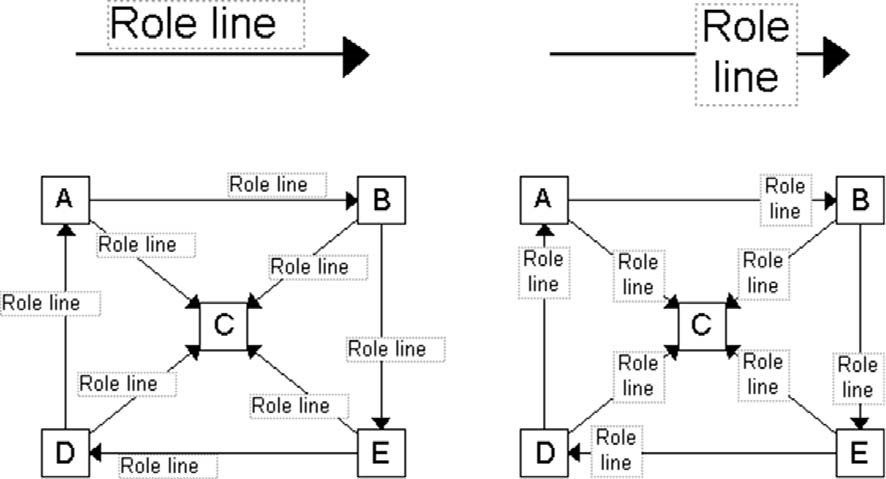


FIGURE 14.2 Role symbol texts

the role line. The offset to the left mentioned above can be seen on the left in the role at A from D. The roles at C from A and E appear to be at the wrong end of the line: the midpoint of the text element is rather far from the role end in the symbol. A better solution overall is on the right, where the text element is square. The text remains symmetrically aligned over the line, and close to the correct role end. This approach will work even in tools that rotate the text boundary. The downside is that the role line will always cross the text element, unlike AB and ED with the previous symbol.

Role Line Routing An important part of the tool support for a graphical modeling language is how role lines connect to objects. A role line must follow a certain route, with the last leg pointing to a certain point in an object symbol, and stopping where it intersects the symbol.

The route of a role line is generally made up of straight lines between freely positioned points. Most commonly, the line will simply head directly from the relationship to the object with no intermediate points. This is the optimal case, as the eye can follow it most simply to read models: the line is the shortest distance between the objects it connects, and thus lies on the natural path of the eye between those objects.

Those with a background in chip design will tend to look for a perpendicular or Manhattan routing of lines: all segments of the line are either horizontal or vertical, with no diagonal segments. While this is harder to draw and to read, and is based on restrictions in electronic design that are not present in modeling languages, it may be hard to persuade users used to Manhattan routing to adopt a simpler approach.

A similar case is the use of curves for roles instead of lines. For some reason this is generally combined with a dotted or dashed line style, making the diagrams even harder to follow. Since editing these curves to look as desired is time consuming, and there is no evidence of any benefit from them, it is probably wisest to avoid them.

It is also possible to define algorithms for automatic routing of role lines, optionally also with layout of objects. A good tool will include some automatic algorithms, although these remain a challenging issue: since the tool cannot know the modeling language, the algorithm cannot take into account special requirements or semantics of different kinds of objects and relationships. DSM models differ in this from simple generic modeling languages, for example, where a single kind of relationship forms a simple tree structure. Any decent algorithm is sufficiently complicated that writing a bespoke version for a DSM language is a major undertaking, and the results are unlikely to be reliably good. Short of bespoke algorithms, the best results so far have been achieved with configurable hybrid algorithms: the user can choose certain kinds of objects and relationships to be routed one way, for example, as a Manhattan tree, and others to be routed by a more generic least-line-length approach.

Connection Boundaries for Role Lines Whatever the route, the last leg of a role line will point toward a certain point on an object. Normally this should be the center of the object, since that is clearest for the eye to read, as well as being esthetically balanced. It also means that the modeler need do no extra work to keep separate the intersections of roles from various sources. As he moves connected objects around, their role lines automatically follow, pointing at the center all the time. In cases where the visual center of an object is not at the center of the total area of its elements, it should be possible to define a specific target point that role lines head to. For instance, in a Use Case Actor symbol you may want roles to connect either to the stick figure, and move the target point up there, or to the text label below, and move the target point down to the center of the label.

“Version 1.0” tools may not offer true center-based role directing, since it requires somewhat more work in calculating the intersection of the role line and the symbol. Instead, they may offer only a limited number of points, normally the midpoints of edges and/or corners of the enclosing rectangle of the symbol. All roles must connect to one of these predefined points, with less readable and esthetic results. The limited range of connection points also forces symbol designers to make sure those points are on the boundary of the symbol, otherwise role lines will stop somewhere in empty space before the symbol, for example, a horizontal line to the middle of an hourglass shape.

Rather than allowing a few fixed connection points, a tool should allow the user to specify a connection boundary around a symbol, at which incoming role lines will stop. In the simplest of cases, where a symbol consists of a single graphical element, that element is itself the connection boundary. Once there are more elements, defining a connection boundary becomes something best left to the metamodeler, although reasonable defaults should be provided. The boundary will generally be a polygon that follows the outline of the outermost edges of symbol elements. Incoming role lines will calculate the intersection point with the first edge of the polygon that they meet, and stop there.

More complicated cases occur where a symbol contains explicit ports, represented by visible nodes on the boundary of the symbol. When connecting to a port, the modeler must normally explicitly choose which port, for example, by clicking that area of the symbol in the model. Since the port is not normally a point, but a small vector graphics shape of its own, it too needs a way to specify where role lines head to and where they stop. Since the role lines cannot head to the object center—they would miss the port—the port should contain its own target point. Rather than draw one complicated connection boundary that snakes its way around the ports, it is easier to allow each port to specify its own connection boundary. This also takes care of the case where a line to one port would also first happen to pass through another port’s area: we do not want the line to stop there.

Generated Text Elements Themajorityoftextelementsinamodelinglanguage will be simple references to property values. Sometimes, more flexibility will be needed, and the text content must be built up from more than one part. For instance, some labels show text in brackets, like UML’s “{abstract}.” If the text in brackets can be of different lengths,eventhis simplest case cannot be handled by a couple of bracket characters as fixed text elements either side of the property.

A reasonable subset of such cases can be dealt with by the ability to specify a sequence of properties and fixed strings. Inventing a new mini language or GUI to specify such sequences should not, however, be necessary. Instead, it is better that the tool simply uses its existing generator language. The language is designed for navigating models and outputting text, which is exactly what this task calls for. If the language is at a high level of abstraction, even a graphic designer who is not a programmer could use it.

Using the generator language also allows us to go further afield in the search for text content to display. For instance, a role may pick up information from its object or port, to show more exactly what the connection is doing. Since the context of the whole graph is available while drawing a symbol, we can also use symbols to provide information about the graph as a whole: errors, warnings, metrics, and so on, as mentioned above.

The downside of complicated generators is that there is no way to know when their content may have changed. If the generator can read any information from this graph, and possibly any subgraph or supergraph, the content could change after almost any operation. If the generator language is powerful enough to pick up representational information (e.g., object positions) or external information (e.g., the results of a system call), there is no hope. A spreadsheet may be simple enough that a smart recalculate algorithm can cope with most cases, but a DSM tool is more complicated. There is no major problem with this: most often the display will be updated anyway, since most operations that affect it will be to closely related elements. In other cases, a simple “Refresh” key will suffice: users are accustomed to programs needing a little nudge in these areas. If the content is valuable, it should not be thrown out because the world is imperfect. Conversely, if the 100% accuracy of the content at all times is vital, we can simply move the generation to a regular generator that is run on demand to produce its output in a separate window.

Conditional Symbol Elements Property values and generated text elements give us most of the dynamic behavior we want in symbols, but in some cases we would like to “generate” something other than text, for example vector graphics elements. This may initially seem to be a task for tools where each symbol is hand programmed, but on closer inspection that is unnecessary. The elements that are to be added are almost always known in advance, so they are not so much generated as displayed conditionally on some value from the model.

Most commonly such elements are small icons or single icon-like characters, displayed based on a property whose value is a Boolean or a selection from a fixed list or other enumeration. It should thus be possible to associate a condition with an element or group of elements, comparing a property to a fixed value. By allowing dynamic information to be displayed graphically rather than textually, conditional symbols can play an important role in creating a truly visual language.

More advanced cases may effectively replace the whole symbol with a different one, depending on the value of a property. This is however somewhat extreme, and normally motivated by problems elsewhere in the modeling language or tool. Rather than effectively merging two types into one, separating them by a new property, it is generally better to keep them as separate types, possibly with a common supertype. The desire for different symbols is probably telling you something about their different semantics, and that will probably later be revealed in differences in rules, properties, and so on.

For individual elements, the conditions may sometimes be more complicated than a single property. As with the contents of text elements, the next step after properties could well be the generator language: that can fetch whatever values are necessary for the condition. On other occasions the value may be from a single property, but the test will not be simple equality with one of a predetermined set of strings. For such cases it is useful that conditions can perform wildcard, regular expression, or numeric comparisons.

14.3.3 Generators

Section 11.2 already looked at different types of generator facilities and many of the features that are desirable in such tools and languages. Here we shall briefly recap the maindesirablepointsofthelanguages,andconcentrateonthetoolsupportfortheiruse.

High Level of Abstraction The language for defining generators should be at a high level of abstraction. While building a generator, the developer has to cope with the language of the models, as well as the language he is trying to output. Both of these take their own slice out of the available brain power, so the less required by the generator language the better.

If the syntax of the generator language is similar to the syntax of the output, it becomes difficult for the brain to separate fixed text parts from the generator language—even with good syntax highlighting. While there is obviously some benefit to using a familiar language, the worst possible case is when the language being generated is the same as the language used to generate it. Such a situation often leads to “metalevel blindness”, or in this case deafness: as we try to understand a program, we often read it to ourselves, and the lack of verbal cues to separate generator from generated easily leads to errors.

Concise, Powerful Navigation A large part of the task of a generator is to traverse the models, so the generator language must have concise yet powerful constructs for model navigation. All too often we see languages with virtually no navigation constructs, leaving the task of navigation to functions generated in each object type. For example, a Button object type may be given a function “triggeredTransitions(),” which answers all the transitions that are connected to this button. This will work fine in a single metamodel, in a single model, where each Button is used in only one graph. However, if the Button is reused in several graphs, or worse in several metamodels, there is no way there can be a single answer to that function: which transitions are triggered depends on which graph we are in, and in some types of graph Buttons may not be used at all for triggering transitions. Lack of language-level navigation constructs thus kills reuse, and inhibits metamodel evolution.

With language-level navigation, the context of the call is known at runtime— including which graph and object we are in. A command like “do >Transitions” will effectively evaluate to something like “currentGraph.relationships(currentObject, Transition).” (See “Reuse of Objects in Models” in Section 14.3.1 for more details on the underlying issues.)

Many-to-Many Mapping Between Graphs, Generators and Output

Files A “version 1.0” tool will generally assume that a single model or single generator maps to a single output file. The first assumption drags the modeling language down toward the abstraction level of the implementation, forcing the same division of information as there. While partial classes or preprocessor include statements may help in some cases, it is much better if the generator facilities do not impose their ownview on the relationship between the structures of models, generators, and output. A single generator should be able to output multiple files, a single file should be able to be built from information spread across multiple models, and so on.

Output Filtering After navigation, the second main task of the generator facility is output of information from models. If the information is already exactly in the format desired, that will be simple. Often, however, this would require modelers to know and remember exactly how a given string in a model will be used in the generated output: if it becomes a variable name, no spaces will be allowed; if it will be in an XML file, ampersands must be replaced with the &amp; entity. Rather than burden the modeler and force models to be less readable, it is better that the generator language offers good facilities for translating output.

Generic functions like “toUpperCase(char \*)” may be useful, but there will always be new cases and exceptions. Within a given DSM solution, certain of these translators will be used many times, so having a shorter syntax will also be useful. A good solution is to allow generators to specify reusable translators in a declarative fashion. It should be possible to use a translator while outputting just a single piece of model information, or to be able to turn them on for a longer block of the generator containing multiple commands.

Syntax Highlighting Most developers have come to expect syntax highlighting from their IDE. In generators this is even more important, as they are a mixture of three syntaxes: the generator language, modeling language, and output language. The generator language and modeling language can be highlighted easily, and at least a single color format can be applied to all fixed text elements—effectively the output language. In all but the simplest template cases, the language in which output is generated will be so split up in the generator as to make parsing it largely useless.

Currently, all generator languages separate fixed text elements from generator language elements with some character sequence. As that sequence will be used frequently, it is useful to keep it as short and unobtrusive as possible: a quote mark is faster to type and read than a template scripting style sequence like <% ¼. It would be interesting to experiment with an approach where the separation was achieved purely with formatting, for example a gray background versus a white background.

Metamodel Integration After the generator language itself, the main language visible in a generator is the modeling language. The generator editor should therefore allow easy navigation and selection of the concepts of the modeling language and their properties. In simple cases it is possible to update generators automatically when the names of modeling language concepts change.

The generators themselves will normally be associated with a specific modeling language: they can only sensibly be run on models of that type, and will contain references to the concepts of that modeling language. It would be possible to go further, and allow generators to be associated with individual object, relationship and even property types. In our experience, however, a full generator will normally consist of less than a few dozen subgenerators. As this is approximately equal to the number of concepts in the corresponding modeling language, scattering the parts of the generator across all types will make generator development less manageable.

Tracing from Output to ModelsandGenerators Particularly while building generators, it is useful to be able to track a given piece of output back to the model element and generator command that produced it. In cases where the intention is not to hide generated code from modelers, being able to track back from output to model is even more useful. Rather than littering the output with numerous trace comments, it is much neater to be able to hide these links “behind” the visible text, in a similar way to hyperlinks in HTML.

Debugger As with most development tasks, building a generator is easy until things go wrong. The fact that generators produce copious output makes their behavior particularly visible: in a way the generator works like a huge set of debugging trace output commands. However, in cases where no output is produced where some was expected, having a true debugger is invaluable.

The debugger should take advantage of its knowledge of the task of the generator, providing clear visibility not only of the current generator call stack but also of the position and history of the model navigation, and the current state of the contents of the output or outputs. Normal facilities for stepping through generators, and over or into called subgenerators, quickly answer most questions about why the generation is not working as expected.

Being able to place breakpoints in a generator saves time in getting to the offending point, if that is known. Where the part of the generator is not known, it may be possible to set a breakpoint on a model element: when the generator navigates to that element in the model, a break is triggered.

14.3.4 Supporting the Metamodeler

In “version 1.0” tools the metamodeler is often left to figure things out on his own, and to hand code things the tool builders were unable or unwilling to implement. Here are a few of the things that help make it possible to metamodel efficiently.

StabilityandCompatibility A difficulty with most prototypes or initial versions is their stability. Here we are not referring primarily to the fact that such tools may crash, but to problems encountered when upgrading to the next version. At least so far, such tools have provided poor or nonexistent support for using modeling languages and models built with the previous version. Of course such tools must move forward and correct design or architecture decisions from the initial version, but still it should normally be possible to automatically upgrade old data to the new version. The fact that this has not been so is largely due to the low level of maturity of these tools on our scale of 1–6. Since the metamodeler needs to write extra code by hand, it is effectively impossible to provide automatic updates: data can be updated, but rarely code.

Documentation and Support It should go without saying that tools should be accompanied by good documentation. This should cover the tools themselves and the metamodeling and generator language they use, as well as tutorials to provide an easy path in. For most metamodelers, this will be their first attempt at creating a modeling language, so more background information and explanation will be necessary than for many other products or frameworks.

Good technical support for the DSM environment will be an important consideration for any serious project. In the early days of a tool, the few early adopters will often receive good support directly from the tool developers themselves. Should a tool prove even moderately successful, the increased number of users requiring support will often swamp the developers. It is thus important that systems are in place for providing technical support, training, and consultancy.

Existing Libraries For the new user, an existing library of DSM solutions can be vital to understanding how DSM works and how to do it well. Although by the nature of DSM only small parts of these could be reused unchanged, investigating how a well put together solution works can have a major effect on how quickly a newcomer gets up to speed.

Browsers and Info Tools Whether modeling languages are built as graphical models, forms or a textual language, the metamodeler requires powerful and flexible tools for browsing, searching and navigating the metamodels. In particular, integration between modeling and metamodeling tools is important: most ideas for improving a modeling language come while using it. The metamodeling experience can be significantly improved by good support for navigating from an element in an instance model to the corresponding concept in the modeling language.

On-the-Fly Metamodel Updates As we have mentioned elsewhere, one of the most important features is the ability to update metamodels on the fly and have models update automatically. The improvements this brings can be likened to moving from character-mapped displays to WYSIWYG word processing, or from debugging by inserting trace output statements to live debugging. These kinds of advances are technically challenging to implement, but the improvements in metamodeler productivity and user experience are undeniable. Of course, there are two levels: updates on the fly are important while the metamodeler tries things out, both when building and maintaining the language. When the modeling language is in production, the metamodeler will rarely be editing the language on the fly, but rather working offline from the other modelers, and only releasing a complete and tested set of changes as an update to the modelers. The models must then update to match the new metamodel, but that is a slightly different issue from the instant feedback to the metamodeler of on-the-fly changes.

AutomaticIcons,Palette,Menus,andDialogs In early versions of tools, the metamodeler is often required to specify by hand the icons, type palette, and sometimes even menus for the resulting modeling tool. This is a clear case of duplication of data and effort: the metamodeler has already specified the object types, their symbols, and which object types and operations are allowed in a given modeling language.

The manual creation of bitmap icons for lists and palettes is particularly time consuming: such icons can be produced automatically from the symbol definitions with appropriate filtering algorithms. This also removes the problem of producing several different sizes of icons for different uses, platforms, or screen settings. Interestingly, today’s graphical designers use this approach when building fixed-size bitmap icons: the icon is drawn as a vector graphic, and automatically scaled to the variety of sizes stored in a multibitmap icon file.

Generating all of these elements automatically ensures that all parts of the modeling tool remain synchronized with the definition of the modeling language. More importantly, this approach significantly reduces the work required of the metamodeler: the task is now simply defining the modeling language, as it should be, rather than building tool support for it. The metamodeler is the expert in the domain, but the creators of the DSM environment are the experts in modeling tool behavior. Any flaws in the behavior should be addressed by the creators of the DSM environment, helping all metamodelers, rather than separately by each metamodeler.

Automatic Property Dialog, Custom Layouts The definitions of the properties associated with a concept should be sufficient to enable automatic creation of proper forms or property dialogs for editing instances of this concept. A true form with the normal data entry widgets appropriate for each property’s data type is the ideal format for editing property values.

Simple table-like property sheets can only handle short, simplevalues well, and are best saved for a quick read-only display of the current element’s values. Where such sheets are constantly visible, their size and font size will always be a compromise between the usability of the sheet and the space left for modeling. Tiny fonts and short space for values may be acceptable for read-only display, but not for editing values. As user feedback is strongly in favor of proper forms with standard data entry widgets, the property sheets are presumably just a quick, cheap solution for the initial version of a DSM environment. The true costs of this saving are however passed on to the modelers, or then to the metamodelers who have to program a proper form by hand.

In-place editing of property values in symbols is an area that requires more investigation. DSM languages strive to avoid representing everything as simple text: if the entirety of a model is seen as text values in its diagrams, there seems little benefit to using a graphical representation. As the values within an object should remain mutually consistent, only editing one value while many are not visible tends to be a poor solution. The user interface for choosing to edit a property value in place, as opposed to simply selecting an object, also frequently causes headaches for both tool developers and users. The display and behavior of the value while it is edited are similarly problematic. Only if these issues can be overcome will in-place editing be a useful addition to true property dialogs.

Automatic Language Help Particularly for new users of a modeling language, good documentation of the language is essential. While nothing can beat or automate the creation of well-written introductory material, a tool can at least provide a textual, human-readable representation of the concepts and rules of the defined modeling language. The metamodeling language and tools should offer fields to add free-form textual documentation of the various concepts, to be incorporated into this automatic language documentation. Keeping this information with the definition of each concept gives the best chance that it will be kept up to date: unlikely if the description is written in a separate file.

14.3.5 Generic Modeling Tool Functionality

Unsurprisingly, the level of modeling tool functionality provided by most initial releases of DSM environments is modest at best. With research tools, this is as it should be: efforts should be focused on new, innovative ways of defining modeling languages, and even empirical trials will focus on real users working as metamodelers, not modelers. For tools intended for actual use in modeling, deficiencies in this area are however serious. Unfortunately, the consequences of these failings reflect poorly not only on the tool itself but also on DSM as a whole. As there are many modelers for each metamodeler, the majority of experiences of DSM across the industry will be based on the modeling tools.

Basic Functionality To be considered for real use, tools must offer at least the basic functionality expected of any program these days, regardless of whether it is a graphical editor, email software, or word processor. These functions include multilevel undo and redo; cut, copy, and paste; direct manipulation of editable elements; and printing and exporting to various file formats.

When copying and pasting, there should also be the possibility to affect whether a direct reference or new copy is created. With complex objects, it must also be possible to specify how deeply contained elements should be copied: objects in properties, subgraphs, and so on.

Direct manipulation is particularly important for laying out diagrams so that they are easy to read and work with. With today’s computers capable of billions of operations a second, it seems odd that so many tools fail to make relationships and role lines follow objects visually as they are moved or scaled.

Multiple Models For some reason, initial versions of tools often omit the ability to work on multiple models (graphs, diagrams, etc.) simultaneously. In a text editor, this may not be so important, as links between files are formed by typing the same sequence of characters as appears in the other file—for example, a function name. In modeling, one of the major benefits is that such links are now made by direct reference, ensuring that links are always synchronized. Tools that fail to support multiple simultaneous models must either fall back on error-prone string references, omit linking altogether, or make significant architectural changes to enable true linking in subsequent versions.

Multiple Modeling Languages Integration between models is not just between models of the same modeling language. Even early languages like Structured Analysis used multiple interlinked modeling languages (Gane and Sarson, 1979), and the more unwieldy of today’s generic modeling languages may have over a dozen different diagram types. Modeling tools must thus support the simultaneous, integrated editing of multiple models of multiple modeling languages.

Multiple Users For all the research on collaboration and groupwork in the 1990s, surprisingly few tools support multiple simultaneous users. As real-time communication and collaboration has spread to the general populace through chat, online games and Internet video and phone calls, software developers have been left like the proverbial cobbler’s children.

Allowing multiple simultaneous users toedit the same diagram seems unnecessary: attempts at this in the 1990s revealed more problems than benefits. It is however useful to allow multiple users to update the information contained in multiple distinct objects within that diagram: the objects may be reused in several places, and edited from there. Everyone should be able to see the latest version of a diagram and reuse its elements, but only one user should be able to edit it.

When designing the meta-metamodel, or to be more exact the data structures that will be used for storing model data, the creators of the DSM environment must consider multiuser issues. They must choose a meta-metamodel that avoids creating hot spots: structures that become a target for parallel updates by multiple users. Where that is unavoidable, they must implement data structures that are appropriate for expected patterns of multiuser access (Kelly, 1998).

Browsers Information in a modeling tool forms part of a relatively complex structure, as with source code. In contrast with source code, much of the structure is made explicit in the internal data structures of the modeling tool, rather than having to be inferred by finding the same character sequence in multiple places. This makes it relatively easy to provide a variety of browsers showing the interrelations of models and their elements.

Elements can be divided by several hierarchies, dimensions, or link structures, for example,

. their types

. the projects they are defined in

. the graphs they are used in

. the hierarchy of graphs

. the hierarchy of objects used as properties

All of these structures can serve as a useful basis for browsing existing objects to examine the state of an application or look for reusable elements. A tool should thus offer browsers supporting hierarchical display along these dimensions, with filtering by project, graph, or type. As these browsers can be built independently of a particular modeling language, they form a good example of the kind of generic modeling tool functionality to be expected from a DSM environment.

Documentation Generation In many senses, a domain-specific model is its own documentation, and the modeling tool is the best way to view the documentation. However, not all users will have access to the modeling tool or be familiar with it, and existing organizational practices probably mandate the use of a particular kind of documentation such as HTML or word processor documents. It is thus useful for a modeling tool to offer export of a complete set of models as documentation. This is discussed in more detail in Section 11.3.3; here it suffices to say that a DSM environment should offer a good generic format for such documentation. Rather than hard coding the function, it should be provided through the tool’s generator facilities, allowing customization for the requirements of a particular DSM solution.

User Documentation and Support As with the metamodeling tools, the modeling tools must also be documented and supported. The users of the modeling tool will normally be unable to separate issues caused by the particular DSM solution from those common to all tools built with that DSM environment. Bug reports and technical support requests must thus often be directed through the DSM team. The quality and maturity of the generic modeling tool will determine whether this is a burden which that team is willing to bear.

14.3.6 Tool Integration

Compared to traditional modeling tools, DSM tools play a larger role in the toolset of the developer: since full code is normally generated, developers often need no longer interact directly with an IDE or compiler. Unlike the dead end of round-trip engineering faced by modeling tools working at the same level of abstraction as the code, DSM keeps code and models clearly separate yet in step. With DSM, each code file is either fully generated—in which case it is guaranteed to be synchronized and yet can also be thrown away at will—or fully handwritten, in which case there will be minimal coupling with the models.

Standalone versus IDE Integration The need for integration with an IDE is debatable: a recent independent survey of lead developers in Europe revealed that only 7% considered Eclipse integration an important feature for a modeling tool, and only 4% wanted VisualStudio integration.

As is often the case, we find good predictions from the move from assembly language to third generation languages. Just as compilers do not fill today’s IDE projects with the assembly files generated from higher-level source code, so it is to be expected that the code produced from a DSM tool will be treated as an intermediate by-product, not a central component. In truly model-driven development, the development environments for models will be focused on modeling, with support for examining code either secondary or delegated to dedicated code IDEs. Unwieldy project trees will be replaced with more visual graphical modeling languages, tailored to the needs of the particular domain.

Call External Tools The integration of DSM tools with other tools and information sources and sinks in the development process thus tends to be one where the DSM tool is calling the shots. A common pattern is for the tool to generate source code files, invoke the compiler, and start up the resulting application for testing (see Section 11.4.1). The most important integration feature in a DSM tool is thus its ability to script the invocation of other tools. The natural medium for this would seem to be a generator language that can produce make files, build scripts, batch files or shell scripts as necessary, and invoke the command processors for these.

Be Called by External Tools Calling external tools will be the norm, but there are also cases where it is useful for other tools to invoke the DSM tool. One common case occurs in organizations with an existing practice of automated nightly builds. Since the primary source is now a model, it should be possible to invoke the tool on a particular model, generate code from it, and exit. This kind of integration is relatively simple, requiring only support for appropriate command-line parameters.

A reference in documentation, version control or a bug management system may point to a particular model or part of a model, and being able to start the tool and open the relevant model can save time. For the first such invocation command-line parameters will suffice, but if there is a need to prevent multiple copies of the tool being opened there should be support for targeting an existing running instance.

API Some integration needs go beyond simple sequential invocation to cases where an external program works in tandem with the DSM tool. Common examples include emulators or simulators that run and animate the models from the DSM tool.

To support this, the DSM tool must offer an Application Programmer’s Interface (API), which reveals a set of its functions that can be called externally. In DSM tools integrated in an existing IDE, an SDK offers tighter integration and lower-level access, but only to programs running in the same IDE process. Since external tools will by definition rarely fall into that category, a separate API is necessary here too.

The protocol used for API communication is something of a difficulty. Older tool APIs tended to be programming language or platform specific, reducing the range of tools that could use them. SOAP appears to offer the best solution currently, as support is available for all but the most esoteric platforms and languages. With SOAP, the interface is at a high level of abstraction. A single call is made to initiate a connection, and commands are issued to the remote tool in the same way as to local functions: the SOAP client takes care of the encoding and decoding of the various data types, as well as the details of the communication.

The second important consideration for an API is the method of data access: should the data structures of the meta-metamodel, metamodel, and model be duplicated in the client program, or should access be via commands to be performed in the remote tool. Duplicating data and data structures is time consuming, unsuited to deep, highly interlinked structures found in models, and causes problems in keeping remote data and local copies synchronized. A pure command-based solution requires a way to identify the model elements to be operated on, and a way to pass model elements as the result of a call.

An appropriate solution in many cases is provided by proxy objects, which are handled in the client program as if they were the real model element, but simply pass any operations sent to them back to the remote tool. There is thus no need to duplicate data or data structure definitions, and the issue of identity is encapsulated within the proxy objects. Functions returning model elements actually return proxies, which can be used as parameters in further calls. Functions returning strings or other primitive data types can return the actual string.

Model and Metamodel Interchange A tool should support the import and export of models and metamodels, for exchange of information with either other instances of the tool or other tools. If the format is to be used with other tools it must of course be tractable and documented. The predictable choice would be some form of XML, but other simpler text formats have also been used. For exchanging models and metamodels with another instance of the same tool, a binary format will suffice.

When exporting to another instance, tools at levels 5 and 6 must solve the question of whether to provide the metamodel with a model—tools at earlier levels can do little more than explode if the file to import was made with a sufficiently different version of the metamodel. If the metamodel is included, the export format will be suitable for use with any instance of the tool, even one that has never seen that metamodel before.

Where an external program wants to operate on the majority of a model or set of models, importing a file is probably more efficient than attempting to work through an API. Working on the imported data and exporting it back to the same tool in another form is almost always a bad idea. A more appropriate use of model-to-model transformations is to process the information into the format supported by another tool that will display, analyze or further process it. In these cases the target “model” is generally not a model in our sense of the word, is read-only, and will be discarded after the process is complete.

Exporting a metamodel to a different tool may be useful when configuring a tool that is later to process exported models. Currently, exporting a metamodel from one DSM tool to another has predictable problems with loss of functionality: only the lowest common denominator is supported by systems such as KM3. For transfer between current “version 1.0” tools this is less of a problem: they have little beyond this, and what they have tends to be expressed as code anyway, which could not realistically be transformed. In a situation with several more mature, data-based tools, better results could be expected from bespoke translations between a pair of tools.

Where a tool supports metamodel import from a textual format, interesting possibilities exist for turning models into metamodels. A modeling language and generators can be built that produce this format, and the results can be imported back into the same tool (or another instance). Many “version 1.0” tools indeed use this as their standard way of creating metamodels, although there is normally some special handwritten support in the tool for the modeling language in question. In tools with the power to achieve this without special tricks or hand coding, it provides a rich and fertile ground for investigating different ways of metamodeling: DSM applied back on itself.

14.4 CURRENT TOOLS

A complete review of current tools is beyond the scope of this book. Given the background of the authors, any attempt would be subject to accusations of bias— founded or unfounded! We will thus look only at what we consider to be the top four environments. As a criterion for this selection we require that the tools form a coherent DSM environment, including metamodeling, graphical modeling, and generation.

We will include two established DSM environments, MetaEdit+ and GME, and two “version 1.0” tools: the DSM plug-ins recently released by Microsoft and Eclipse for their IDEs. Within Eclipse there are also some other “version 1.0” or prototype plug-ins that we shall mention briefly.

### CURRENT TOOLS

Q1 14.4.1 MetaEdit+ (MetaCase), 1995–

First released in 1995, MetaEdit+ is a descendant of the earlier MetaEdit in terms of its concepts and development team, but a fresh start in terms of its code. MetaEdit+ aimed to rectify several architectural decisions in MetaEdit that proved to restrict its scalability and efficacy. Some of the decisions had simply been due to limited resources, but others were based on false assumptions: what had seemed a good idea turned out to work poorly in practice. Interestingly, these same assumptions appear to be present in the current crop of “version 1.0” tools: perhaps this is a phase all such tools must go through.

Early development of MetaEdit+ concentrated on laying a strong architectural foundation, based on the experiences with MetaEdit and building on earlier work in QuickSpec and PSL/PSA. Initial development took place in the MetaPHOR research group, which had been formed at the university in 1987. The company, MetaCase, was founded in 1991, and development efforts progressively shifted from the university to the company. This move was completed when the core developers finished their doctorates in 1998. While much has happened since then, the academic history of the early days does mean that there are around 30 theses and 150 articles documenting the initial versions of MetaEdit+ (e.g., Kelly et al., 1996) and the ideas behind it (e.g., Tolvanen, 1998).

MetaEdit+ is at level 6 in our scale of tool evolution: metamodeling and modeling take place in the same integrated environment, running as a single or multiuser system on any of today’s major platforms. With version 4.5 released in 2006, it fulfills all the criteria we have identified above as desirable for a DSM environment—time to identify some new requirements! Humor aside, given the greater time we have had to evolve MetaEdit+, it would be surprising if we had not been able to develop it to fulfill what we see as necessary from our experience and understanding. The more interesting question is in which areas are there holes in our experience or blind spots in our understanding.

According to independent industry experts, such as Scott Ambler (2006) and Markus Vo¨lter (2006), MetaEdit+ is the most mature and sophisticated DSM environment. A recent independent survey in Europe by MediaDev (2006) revealed that it is also the most widely known, recognized by over 50% of lead developers. With customer projects ranging in size up to several hundred modelers and over 10 years of duration, MetaEdit+ is also widely recognized as having the most experience of DSM in use. Even so, we still feel there is much to improve and learn: current DSM practice and tools will face many challenges and choices as adoption spreads.

14.4.2 GME (Vanderbilt University), 2000–

The Generic Modeling Environment (GME) grew like MetaEdit+ out of an extensive background of earlier research and practical experience in metamodeling and modeling tools. The earlier work with MultiGraph Architectures at Vanderbilt had proved the value of DSM, and created a framework for building new modeling tools. Specifying the metamodel in textual files proved time consuming and inhibited the evolution of the modeling languages in use. Nordstrom’s thesis (1999) outlined a meta-metamodel suitable for defining DSM languages in his domain of electrical engineering, and a Generic Modeling Environment that could be configured by files generated from metamodels made with this language.

GME’s meta-metamodel differs significantly from that of other tools. It has a strong concept of port coupled with a weak concept of relationship, probably because of the electrical engineering inheritance: wires are just wires and connect specific pins on components. The tool support also reveals this same balance: all connections between objects are autorouted as uneditable horizontal and vertical lines. There is no separate first-class concept for graph, but objects are allowed to contain other objects to form a hierarchy.

Metamodels are specified in a simple UML-like language, which makes extensive use of stereotypes to distinguish the various metatypes—an odd choice in a DSM environment where different concepts should normally be given different symbols. Constraints are specified in a dialect of OCL. There is no graphical symbol editor and only a simple macro-based system for generating reports: all symbols and most generators are written by hand in C++.

GME is at level 5 in our scale of tool evolution: a metamodel is turned into a binary file, which configures a second instance of GME. Models do not update automatically when the metamodel is updated: an explicit operation can be chosen to try to update them, but this will often be unable to complete the update. A secondary scheme involves exporting the model as XML, upgrading the metamodel, and then reimporting it; even this will fail if the old model is not completely valid in the new metamodel. In that case, the user must return to the older version and correct the model by hand to be valid in both old and new versions of the metamodel—presumably impossible in some cases except by deleting parts of the model—and only then can she update.

GME is only available on Windows. At least earlier, it also had a multiuser version using an ODBC connection to a Microsoft SQL server back end. A database can contain multiple projects, each containing multiple models. The models must all be from the same set of modeling languages. Integration with other handwritten code modules for model analysis or extra tool functionality is via a COM interface, or a C++ interface built on top of COM.

14.4.3 DSL Tools (Microsoft), 2006–

In the Visual Studio 2005 SDK 3.0, Microsoft released the first version of their Domain-Specific Language Tools: a combination of frameworks, languages, editors, generators, and wizards that allow users to specify their own modeling languages and tools. This support for DSM forms the major new element of Microsoft’s wider Software Factories project. Model-driven development has backing from the highest levels of Microsoft, with Bill Gates claiming the approach will be the most important innovation in software over the next 10 years (Seeley, 2004).

As its metamodeling and modeling tools will both only work as part of Visual Studio, DSL Tools are only available on Windows. Worse, the license agreement

### CURRENT TOOLS

(since amended) demanded that no code generated from DSM tools built with DSL Tools may ever be run on a non-Microsoft platform:

Code Generation and Optimization Tools. You and your end users may not use any code generation or optimization tools included in the [Visual Studio SDK] or Visual Studio (such as compilers, linkers, assemblers, runtime code generators, or code generating design and modeling tools) to create programs, object code, libraries, assemblies, or executables to run on a platform other than Microsoft Platforms.

From an early stage, Microsoft decided that MOF and UML were unsuitable for describing modeling languages, stating clear arguments similar to those we have discussed elsewhere. While this could be imagined to be a thrust against the IBM– Eclipse–OMG trinity, many of the leading DSLTools figures were previously closely involved with UML and the OMG. Indeed, the core of the team seems to have been brought in from outside Microsoft: Jack Greenfield was a Chief Architect at Rational, Keith Short was CTO at Texas Instruments, Steve Cook was a Chief Architect at IBM.

The resulting meta-metamodel is not, however, all that different from MOF, and indeed shares many of its failings. There is no clear concept of multiple graphs, no n-ary relationships, roles and ports are not first-class elements, and reuse is hampered by the requirement that each element be a child of exactly one strict aggregation. Ironically, the first example model in the documentation shows a clear n-ary relationship, faked by carefully overlaying four binary relationships in exactly the same place.

On the positive side, the lines in diagrams are recognized as being roles: a line between two objects thus consists of a relationship at the midpoint, one role line to one object and another to the other object. The meta-metamodel clearly separates aggregation or containment from normal relationships. The latter are however stored as facts about the objects involved, effectively preventing transparent object reuse.

Metamodels are specified in a graphical version of this meta-metamodel. Oddly, there is no way to start from an empty metamodel: they are always created from a wizard where the user must choose the closest existing metamodel “template”, and even the smallest already contains concepts. Users must thus delete these or preferably rename them. In an unusual decision, the metamodel is always automatically laid out in a vertical tree. To maintain the pure tree structure, if an element is referred to from more than one place, it must be duplicated as a reference element to elsewhere in the tree. This structure quickly becomes unwieldy, and it is hard to see what benefits the designers expected from this decision.

The concrete syntax is severely limited: symbols may consist of only a single geometrical figure, and the only supported figures are rounded rectangles or their extremeversions, rectangles and ellipses. Bitmap or other external images can be used, but do not scale, and compartment shapes are used to handle aggregations. In a novel decision, the elements representing graphical symbols are drawn in a “sidebar” to the main metamodel, paralleling the elements they represent. Confusingly, the natural representation of these elements is not used in the metamodel: whether it is a red rectangle or a blue ellipse, the element is always shown as a gray rounded rectangle.

DSL Tools are at level 3 in our scale of tool evolution: the metamodel is transformed by generators into C# and possibly C++ code. This code is extended with handwritten code for constraints, proper symbols, and improved editor behavior. The result is then compiled, built and opened as the DSM tool in a separate debugging instance of Visual Studio. When the metamodel is changed, regenerated and recompiled, this instance must be closed and a new instance opened.

Currently, there is little support for updating a modeling language. If the name of a modeling language concept is changed, the corresponding model elements all disappear, effectively being deleted when the model is opened. Similarly, if a property on a relationship in the metamodel is changed, existing relationships may give errors or disappear.

Generation facilities consist of a text template language, mostly consisting of C# or Visual Basic that targets the generated data structures of the modeling language and tool. Templates can read from multiple models, but each template can only produce one file. The template language’s support for modularizing templates is limited to simple include commands.

14.4.4 Eclipse Modeling Project (Eclipse), 2006–

The Eclipse Modeling Framework (EMF) allows the generation of simple tree views and property sheets from metamodels specified in XML files. The Graphical Editing Framework (GEF) is a framework that allows graphical editors to be written using Draw2D for display. EMF and GEF are not integrated and GEF in particular operates at a low level of abstraction, leading to serious inefficiencies if used to create a DSM tool. To address these concerns, the Graphical Modeling Framework was developed. GMF integrated and extended an improved GEF from IBM with GEF code generators from Borland. The underlying models thus remain in EMF, but GMF subsumes and encloses GEF to provide the representational information and diagram editor.

GMF 1.0 was released in June 2006, as part of the Callisto release of Eclipse projects. The current 1.0 release seems to be regarded as falling somewhat short of what was intended, because of the time constraints of simultaneous release with other Eclipse projects. The direction toward the next version is however correct, with getting it right being preferred over backward compatibility. According to GMF project lead Richard Gronback (2006), “We anticipate a 2.0 release for the next release rather than a 1.1 perhaps because [sic] API breakage that we expect.”

Creating a simple editor with GMF consists of building five different XML files, most of which can be created with graphical or form-based modeling tools:

. Domain model: abstract syntax, Ecore model

. Graphical definition: concrete syntax, XML or tree view and property sheets

. Tooling definition: palette tools and actions, tree view and property sheets . Mapping definition: links first three models, tree view and property sheets

. Generation model: customize plug-in and UI, tree view and property sheets

### SUMMARY

The use of Ecore—effectively MOF—for the abstract syntax leads to the normal difficulties: relationships in the DSM language may be classes or associations in the metamodel, and associations in the metamodel may represent relationships, roles or properties in the DSM language. These ambiguities cause the need for the tooling and mapping definitions, which provide information missing from the Ecore model. Having the information spread across so many models leads to significant duplication of data and effort, with little increase in useful flexibility.

The current releases of EMF and GMF are at level 3 on our scale of tool evolution, at least for real-world DSM languages: level 4 is possible for simpler examples. Many of the identified required features are missing. Providing a more detailed listing for a “version 1.0” tool in a book seems little use: readers should check the state of the current version themselves.

No native code generation facilities are included, but the separate JET project offers simple textual templates. More powerful facilities for generation or model-tomodel transformations can be provided through third party tools or frameworks such as OAW or ATL. As these operate on essentially any XML input, they are of course also usable with a wide array of other tools.

Other Eclipse-based tools exist, with the largest growth being in tools or frameworks that aim to raise the relatively low level of abstraction of GMF. Two prominent ones, TOPCASED and GEMS, appear to predate GMF, and to have originally been intended to perform the same function as GMF. They have now been retargeted to use GMF, although this should in theory have little effect on their users, if they already hid the low level nature of another graphical editor framework. TOPCASED adds little new and is only at version 0.3.0, so we will look at GEMS.

GEMS is an extension to GMF built by Jules White at Vanderbilt University. It aims to remove some of the low-level work of using GMF, and focuses particularly on adding constraint solving to the resulting editors. Constraints can be expressed in Prolog, giving better expressive power than Java or OCL, and can be used either to provide a simple check of a model or to add new elements to a model automatically, so that the constraint is fulfilled.

A commercial, closed-source Eclipse plug-in with a similar approach to GMF is XMF-Mosaic from Xactium. Again, multiple modeling languages are used to describe a DSM solution. In this case, there are also languages for specifying code generation: modeling language concepts are mapped to simple object-oriented programming concepts like classes or methods, and separate generic mappings are provided from these concepts to specific languages like Java.

14.5 SUMMARY

DSM environments are one of the most conceptually complex types of software. When made well, they can allow one of the greatest ranges of behavior for relatively simple input. Turning such complexity and power into apparent simplicity is however a difficult task. Building the foundation of such an environment without significant experience in DSM is possible, but such foundations rarely have the necessary qualities of stability and flexibility. Lack of these qualities will be most visible as use of the environment grows in time and size. On the time dimension, the evolution of metamodels—and the parallel evolution of the environment itself—will be slowed and soon stopped by the weight of unknown and unintended consequences of changes. On the size dimension, the tool will face difficulties in scaling to support multiple models, modeling languages, and users.

Most DSM environment developers have consistently shown a disregard for earlier work. Of the tools whose first version was released in 2006, not one matches even the power and simplicity of GraphTalk, released in 1988, let alone that of mature tools such as MetaEdit+ and GME. Paradoxically, it was the success of UML that rehabilitated modeling after the disillusionment caused by the failure of CASE, and yet that same success was instrumental in the decline and fall of many promising metamodeling tools of the 1990s. Dying before Google and the Wayback Machine, their experience is effectively lost to current developers, even if they published widely in journals and conferences.

If there is one area where the new DSM environments must stop and rethink, it is their meta-metamodels. All are strongly influenced by UML Class Diagrams, a language for a domain completely different from that of graphical modeling languages. All suffer heavily from this mismatch. It would be ironic if just at the point where our industry has acknowledged that UML has failed as a way of building applications (as opposed to describing them), and is moving toward DSM as the best way forward, DSM environments were to blinker themselves into offering UML as a way of building modeling applications.