Making model-based code generation work

Full code generation is possible when both the modelling language and generator are fitted to the requirements of one company and domain only. Dr. Juha-Pekka Tolvanen explains how model-based code generation can be achieved.

The promise of modelling has been to shift the focus from implementation to design. Models serve as mechanisms to get a better understanding but they can also be input for code generators. This automates development leading to improved productivity, quality and complexity hiding. Unfortunately, many current modelling languages are based on the code world and offer only modest possibilities to raise design abstraction and to achieve full code generation.

For example, UML uses directly programming concepts (classes, return values, etc.) as modelling constructs. Having a rectangle symbol to illustrate a class in a diagram and then equivalent textual presentation in a programming language does not provide real generation possibilities – the level of abstraction in models and code is the same! As a consequence of this, developers easily find themselves making models that describe behaviour and functionality that they find easier to write directly as code.

Limited code generation possibilities force developers to start manual programming after design. This kills the idea of full code generation and minimizes the productivity benefits. It has also led to round-trip problems: if the developer is code-focused, his models merely visualize static structures. If the developer is modelling-oriented, a lot of effort goes into rewriting generated code and keeping all the other models than class diagrams up-to-date. Having the same information in two places, code and models, is a recipe for trouble.

Generation challenges can be solved in a similar manner as in the past with programming languages: by continuing to raise abstraction. Models should not be conceived to visualize code, but describe higher-level abstractions above programming languages. Similarly, it was better to move to C to raise the abstraction than start visualising Assembler code!

The higher level of abstraction varies between applications and products, though. Every domain contains its own specific concepts and correctness constraints. Therefore, modelling languages need to be specific for each domain. Domain-Specific Modelling (DSM) languages narrow down the design space, often to a single range of products for a single company. Accordingly, models are made up of elements representing things that are part of the domain world, not the code world. Let's take some examples. If we are developing mobile applications for a device, why not have the concepts of the UI and mobile services directly in the design language? It is far more natural to think about application logic with 'List', 'SMS send', 'View' etc. than with C code.

Likewise, while developing voice communication systems, microcontroller concepts like 'Menu', 'Prompt', and 'Voice entry' are closer to the application domain than Assembler mnemonics. Higher-level specification languages use these domain-specific concepts directly as modelling constructs.

Because the language follows domain abstractions and semantics, modelers perceive themselves as working directly with domain concepts. These domain concepts are typically already known and in use, are more natural and reflect already the underlying computational models needed to design the products. Final code (assembler, 3GL, object-oriented etc.) can be still generated from these high-level specifications. Cornerstone for the automated code generation from models is that both the language and generators need fit only company's requirements.

Recently, open and customizable technologies have emerged that allow developers to change both the design languages and/or code generators to meet different requirements of software development. Hence, experienced developers in a company can adapt the design languages and generators to a specific domain, and then actual products can be designed with domain-specific languages and generated directly from models.

Language creation example

Suppose you manufacture digital wristwatches and your developers make the watch applications, such as stopwatch or world time. Before any new features can be implemented developers must design them in the watch domain. This involves applying the terms and rules of the watch, such as buttons, alarms, display icons, states and user's actions. The domain-specific method applies these very same concepts directly in the modelling language. An example of such a modelling language is illustrated in figure 1.

The model represents the time setting feature: the actions a user can make by pressing buttons, the display elements blinking, and the actions changing the time. Accordingly, the definition of the language starts by identifying the terminology and concepts for the modelling language (like button, icon, etc). These are documented formally into a metamodel that can be executed in metamodel-based modelling tools.

In addition to the terminology, the DSM

language follows a computational model that in this case is the state machine – a typical computational model used with embedded software. However, the level the state machine operates on and the rules it applies are not the implementation, but product rules. The model operates on a higher level and thus there is far less need to do modelling than in code visualization focused languages, but still just enough for code generators to produce the required code out of the models.

In the final step of language creation we enrich and narrow the semantics of the state machine to focus on the concept of the watch domain. To use our example there are two watch-specific extensions in our state machine. First, the transitions can be triggered only by the user interaction when a certain button is pressed. Second, the actions taking place during the transition may only operate on time unit entities. Also the set of possible operations is limited: one can only add or subtract time units or roll them up or down. It must be emphasized that if further needs arise in the future, we can simply extend the set of possible operations or define new entity types to operate with. With the above basic operations we can cover in our example all current needs of our watch family and automate development with code generators.

True model-based code generators

The generator specifies how information is extracted from the models and transformed into code. This process depends on and is guided by the modelling language with its concepts, semantics and rules and the input syntax required by the target platform.

To be usable, the generation process must be complete: full code is generated from the application developer's point of view and manual rewriting of the code is not needed. This completeness has been the cornerstone of other successful shifts made with programming languages. Ever seen anybody manually edit Assembler and try to keep their C code in synch with it? Similarly, the generated code should be simply an intermediate by-product on the way to the finished product, like .o files in C compilation.

Such full code generation is difficult, if not impossible, to achieve when the generator (and the modelling language giving the input) is designed to fit to almost all situations. Full code generation, however, is possible if both the language and generators need fit only one company's requirements. This means that the code generator must work well with the modelling language from which it gets input and with the target platform on which the generated code will run.

Metamodels guide the generators

While models provide the data as input for the code generation process, the actual navigation and retrieval of the design information is carried out according to the metamodel. Therefore, the metamodel should conform to a computational model that is natural for the developed product. In most cases certain modifications or extensions for the basic model are needed for domain-specific or code generator purposes. This is done to ensure that the models capture all essential static and behavioural aspects of the product as input for the code generator. For example, in our Watch example, we chose the state machine as our computational model and then enriched it with such concepts as time units to meet the requirements for the domain and code generation.

Role of the platform framework

A platform provides a well-defined set of services for the code generator to interface to: for example, generated code can directly call the platform components and their services. Often, though, it is good to define some extra framework utility code or components to make code generation easier. Such a framework can be used on top of the platform in form of libraries, components and code templates. The framework is not necessarily an extra burden only required by the code generator. Actually, in most cases the underlying software architecture already utilizes various libraries, components or other reusable parts that can also support code generation.

The key issue in building a code generator is how the models' concepts are mapped to code. The output is not defined as concrete code but more as an example or template. In the simplest cases, each modelling symbol produces certain fixed code, including the values entered into the symbol as arguments. The generator can also generate different code depending on the values in the symbol, the relationships it has with other symbols, or other information in the model.

The generator definition should be kept as straightforward and simple as possible. This can be achieved by not dealing with variation or low-level implementation issues within the generator. The framework and component library can make this task easier by raising the level of abstraction on the code side.

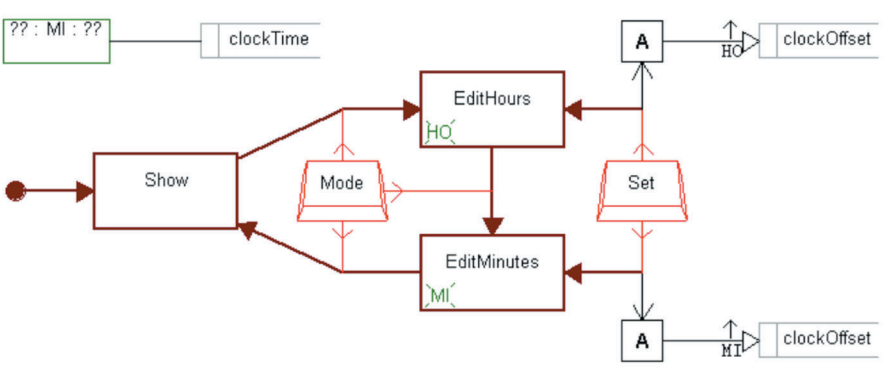
Also having domain-specific models with correctness constraints available makes generator definition easier: generator does not need to check that input is correct. Proper modularization and reuse help a lot when building the code generator. For example, if you can isolate variation handling for different target platforms into own modules, it becomes very simple to widen the platform support as adding new platform requires only new versions of those modules, not the whole generator.

Example of generator

So far we have been discussing the different parts of the code generation solution. The next question of course is, how to put them together to achieve the 100% code generation. To understand this, consider the listing 1, which is the generated code for the state machine illustrated in figure 1.

First thing we see from listing 1 is that the complexity of the state machine is hidden from the generator by implementing its elementary behaviour as an abstract framework class. The concrete state machine is then sub-classed from this abstract class (line 5) and initialized with data from design models in class constructor (lines 13

Fig 1: State machine with watch domain extensions



Listing 1: Code generated from state diagram illustrated in figure 1.

```
01 // All this code is generated directly from the model.
02 // Since no manual coding or editing is needed, it is
03 // not intended to be particularly human-readable
04
05 public class SimpleTime extends AbstractWatchApplication {
06
07  // define unique numbers for each Action (a...) and DisplayFn (d...)
08  static final int a22_1405        = +1; //+1+1
09  static final int a22_2926        = +1+1; //+1
10  static final int d22_977         = +1+1+1; //
11
12
13  public SimpleTime(Master master) {
14   super(master);
15
16   // Transitions and their triggering buttons and actions
17   // Arguments: From State, Button, Action, To State
18   addTransition ("Start [Watch]", "", 0, "Show");
19   addTransition ("Show", "Mode", 0, "EditHours");
20   addTransition ("EditHours", "Set", a22_2926, "EditHours");
21   addTransition ("EditHours", "Mode", 0, "EditMinutes");
22   addTransition ("EditMinutes", "Set", a22_1405, "EditMinutes");
23   addTransition ("EditMinutes", "Mode", 0, "Show");
24
25   // What to display in each state
26   // Arguments: State, blinking unit, central unit, DisplayFn
27   addStateDisplay("Show", -1, METime.MINUTE, d22_977);
28   addStateDisplay("EditHours", METime.HOUR_OF_DAY, METime.MINUTE,
d22_977);
29   addStateDisplay("EditMinutes", METime.MINUTE, METime.MINUTE, d22_977);
30  };
31
32  // Actions (return null) and DisplayFns (return time)
33  public Object perform(int methodId)
34  {
35   switch (methodId) {
36    case a22_2926:
37     getclockOffset().roll(METime.HOUR_OF_DAY, true, displayTime());
38     return null;
39    case a22_1405:
40     getclockOffset().roll(METime.MINUTE, true, displayTime());
41     return null;
42    case d22_977:
43     return getclockTime();
44   }
45   return null;
46  }
47 }
```

- 30). This is an example of how to implement a framework counterpart for a logical model construct.

Lines 33 – 46 are an example of code generated for behavioural aspects of the application. For each state transition, there is usually as set of actions that take place during the transition. In our Watch language these actions are limited to cover only basic time unit arithmetic. This leaves us with three operations to deal with: plus, minus and rolling. These operations are implemented as primitive services in our framework. When such a service is needed, the code generator only produces a call for it (for example, as in line 37 or line 40).

As already mentioned, domain-specific models describe the application functionality in code-independent manner at a higher level of abstraction. Therefore, the same models let us generate code for multiple platforms. For example, C code could be generated from the same designs: only the generator is different, not the application designs.

Domain-specific modelling allows faster development, based on models of the product rather than on models of the code. Our example above gives one illustration. Industrial experiences of DSM show major improvements in productivity, lower development costs and better quality. For example, Nokia reports that it now develops mobile phones up to 10 times faster in this way, and Lucent - that it improves their productivity by 3-10 times depending on the product. The key factors contributing to this are:

- The problem is solved only once at a high level of abstraction and the final code is generated straight from this solution.
- The developers' focus shifts from the code to the design, the problem itself. Complexity and implementation details can be hidden, and already familiar terminology is emphasized.
- Consistency of products and lower error-rates are achieved thanks to the better uniformity of the development environment and reduced switching between the levels of design and implementation.
- The domain knowledge is made explicit for the development team, being captured in the modelling language and its tool support.

Implementation of domain-specific modelling and code generation is not an extra investment if you consider the whole cycle from initial design to working code. Rather, it saves development resources: traditionally all developers work with the domain concepts and map them to the implementation concepts manually. And among developers, there are big differences. Some are experts, but most are not. So let the experienced developers define the concepts and mapping once, and others need not do it again. A code generator that is defined by an expert will, no doubt, produce applications of better quality than those created manually by average developers.

Dr. Juha-Pekka Tolvanen is the CEO of MetaCase. He has acted as a consultant world-wide for method development and he has published papers on method engineering in several journals and conferences. He can be reached at jpt@metacase.com.