RESOURCE GUIDE

Modeling for full code generation

By Angelo Hulshout and Juha-Pekka Tolvanen, PhD

Domain-Specific Modeling (DSM) is an approach for attaining higher software development productivity based on the notion that significant increases in the level of abstraction developers work with achieve the most significant productivity improvements. Two key examples of this include the moves from punch cards to assembler in the 1950s and from assembler to FORTRAN a decade later. In this case study, Angelo and Juha-Pekka explore an application of DSM and discuss the software development improvements achievable through this approach.

DSM overview

DSM is a modeling approach that raises the level of abstraction developers work with by specifying the solution using concepts of the Application Domain. This is achieved through a DSM language that follows domain abstraction semantics, allowing developers to focus on the solution rather than the technical implementation of the solution. Often, these technical implementations can be generated directly from the domain-specific models using a dedicated code generator.

This automation is possible because of domain specificity; both the modeling language and code generators fit the requirements of only a single narrow domain. This can be the domain of a single company or solution provider or a domain extensively covered by standards applied by multiple competing and cooperating companies. Standards such as AutoSAR in the automotive domain and industry efforts such as IP Multimedia Subsystem (IMS) in the telecommunications and networking domain provide great opportunities to increase developer productivity and product quality through DSM.

Telecommunications and networking is the domain of our example case, which will be presented first, followed by a discussion of general issues related to DSM.

The case

The features and services in the telecommunications industry are what phone exchanges and VoIP servers are dealing with at present. Examples of basic features include call forwarding, connect-to-voicemail, and call-back-on-busy. Current efforts toward realizing IMS involve integration of these traditional telecommunication services with the Internet-enabled services such as instant messaging and multimedia communications.

All this functionality must be supported by the machinery and software that forms the telephony network, including phones, exchanges, and computers. Software and test engineers provide the core functionality of this material, but the burden of making it work for end customers lies with telephony providers' service engineers and systems managers. Often, their job is a matter of manually tweaking configuration data and call scripts, a repetitive and ever-more error-prone, manual task.

To facilitate this task while reducing the amount of work and possible errors, we developed a solution based on DSM.

From concepts to language

Before defining a domain-specific language, the domain itself must be defined. In IMS based on our architectural design, we have identified three layered domains (Figure 1). The bottom layer is the Connectivity Domain, which consists of the existing worldwide communications infrastructure: the Internet, wireless networks, fixed-line telephony networks, and so on. The middle layer is the Service Enabling Domain, where the aforementioned features and services are found. Finally, the Application Domain is where end users can access services and where service engineers and technical product managers must set up these end-user services. Therefore, our DSM efforts will focus on this domain.

In the architecture defined for our IMS implementation, the Application Domain layer is populated by *steps*. These steps represent the smallest possible unit of execution in the Application Domain (Figure 2). Examples of steps are *actions* (for example, set up connection), *functions* (for example, transform phone number), and *decisions* (for example, receiver present). All end-user services can be built using these small steps, but because of their granularity, configuring the steps individually is a tedious job.

What makes it even more difficult for our intended service creation users is the fact that steps are connected in flows, but they relate to events handled by various processes (tasks) in the Service Enabling Domain. The differences between the flow-driven Application Domain and the event-driven world underneath is difficult to grasp.

To make things easier, we raise the abstraction one more level by combining steps into larger blocks called *Service Building Blocks* (SBBs). As seen in Figure 3, these represent services in the Application Domain that can simply be connected in a flow model without adding intermediate decisions, checks, translation, and so on.

Concepts

Based on this reasoning, our language will consist of different SBBs, relations between them, and some additional concepts to make flow modeling possible. This results in the following list of items used to model the flow of establishing a connection between a calling party (the caller) and the destination the caller wants to connect to. Graphically, we adopt a flowchart-like approach.

List of concepts

- **Caller:** Despite the extent of IMS services, we still deal with callers as have existed in telephony for years.
- **Destination:** The destination being called by the caller or an alternative destination determined by the IMS system.
- **SBB:** A service consisting of multiple steps intended to deliver a certain service to the end user. In reality, we have multiple specific SBBs available, and more may be added in the future.
- **Terminate:** An SBB that terminates the connection for reasons to be defined in the model where it is used.

Relationship between concepts

- **Interlink:** Connection between SBBs.
- **Enter:** The relation between caller and the first SBB encountered.
- **Exit:** The relation between caller and the last SBB encountered.
- **Announcement:** Can be used on any of these relations to play an announcement to the caller.

Each of these concepts has various properties. The most important properties are for concepts that have multiple types, such as switch, and address, for concepts that influence or use addresses. Figure 4 shows the graphical symbols used for the concepts and relations listed using a small sample set of SBBs.

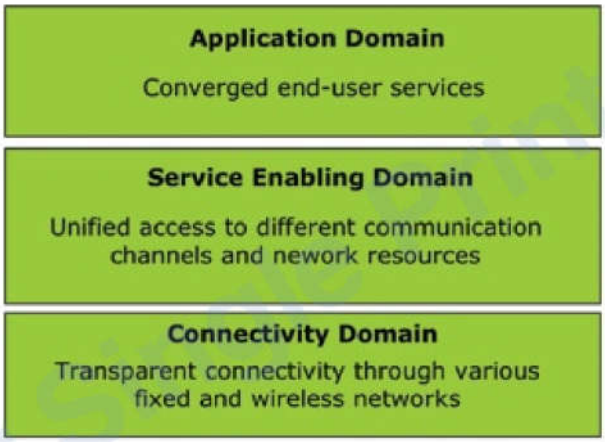

Figure 1

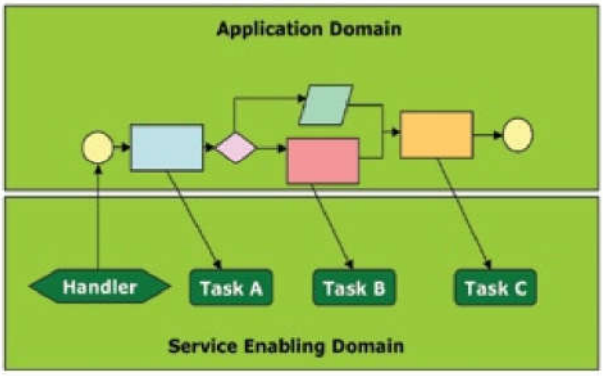

Figure 2

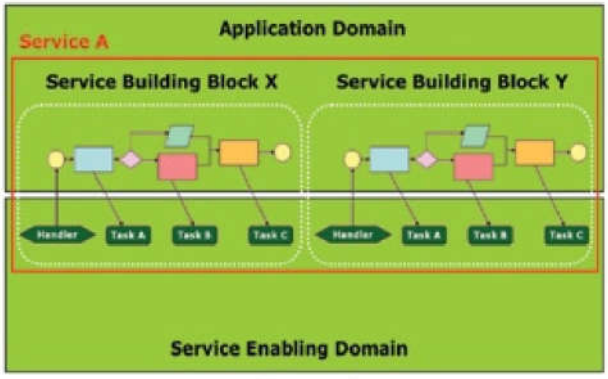

Figure 3

Rules and constraints

In addition to pure modeling concepts we also must identify a set of rules and constraints. These control languages use and enforce model correctness. For instance, we cannot have more than one initiating party for a single call; even in conference calls someone has to make the first move. We are creating modeling languages suitable for configuring and scripting the services, so we define our language such that only one caller can be specified. Also, at least one destination or terminate must be present in each model to have a complete flow. Other constraints may be related to the order in which SBBs are connected, depending on their function.

A sample model

Using our now-defined language, we can draw a sample model that depicts its usage. Figure 5 shows how a script for *presence-based routing* is implemented. Users in IMS can specify whether or not they are present or want to be called at a specified location (for example, their office terminal). In this case, we want to redirect the call to an alternative location, the destination person's cell phone, after playing a notification message to the caller. If the contact has turned off presence at the alternate location as well, the call will be terminated after a standard message is played to the caller.

A service engineer allowed to model this service in this way will not be able to make many mistakes; the tool will prohibit breaking constraints. This engineer does not need to be bothered by the details of the steps or the event-driven nature of the Service Enabling Domain. Also, the customer will be able to discuss desired functionality with the service engineer on the end-user application level.

Generating code and data

Of course, to make this DSM approach truly useful, we must generate the actual scripts that the IMS system can understand and that the service engineer would otherwise have to painstakingly create manually. Figure 6 depicts a more technical overview of the IMS system's realization of the Application Domain and Service Enabling Domain. The Service Enabling Domain is based on an application server, which combines steps, coded in software, according to XML configuration files.

These configuration files are generated from our models, an example of which can be seen in Figure 7. Next to the XML files, we can also generate end-user and service documentation from the models based on a description property that is part of each element in our modeling language.

A code generator specifies how information is extracted from the models and transformed into code. This process depends on the modeling language, as the models that are made form the input to the generator. The key issue in building a code generator is how the model's concepts are mapped to code. The domain framework and other available libraries can make this task easier by raising the level of abstraction on the code side. In the simplest cases, each modeling symbol produces certain fixed code, including the values entered into the symbol as arguments. The generator also can generate different code depending on the values in the symbol, the relationships it has with other symbols, or other information in the model.

To produce the XML call scripts for IMS, the generator simply must traverse the model elements along the directed relations and fill in the necessary nested elements.

Future options

As our IMS solution is still very young, many possible improvements and changes can be identified. With respect to what is presented in this article, one topic being considered is whether or not it is useful to model SBB internals using DSM as well. Currently, steps are hand coded, and combining them into SBBs is partly done manually (Java code) and partly by the code generator (generated XML). A trap to avoid is so-called visual programming: developers model their solution on an abstraction level that is so close to code that the added value of DSM is lost. In essence, DSM helps generate solutions in such a way that if we test one and it succeeds, all others will work as well. With visual programming, we would need to debug and test each solution separately.

Why DSM?

DSM offers significant advantages over traditional manual approaches. Perhaps the most recognizable benefit is the increase in development productivity, a

A service engineer allowed to model this service in this way will not be able to make many mistakes; the tool will prohibit breaking constraints.

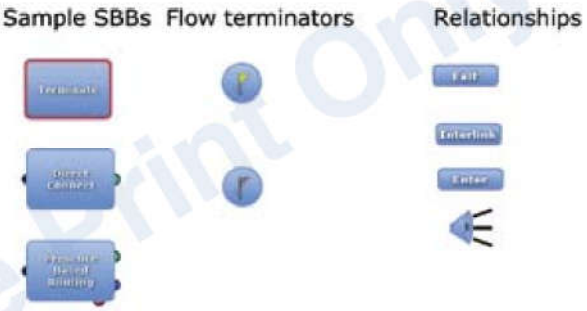

Figure 4

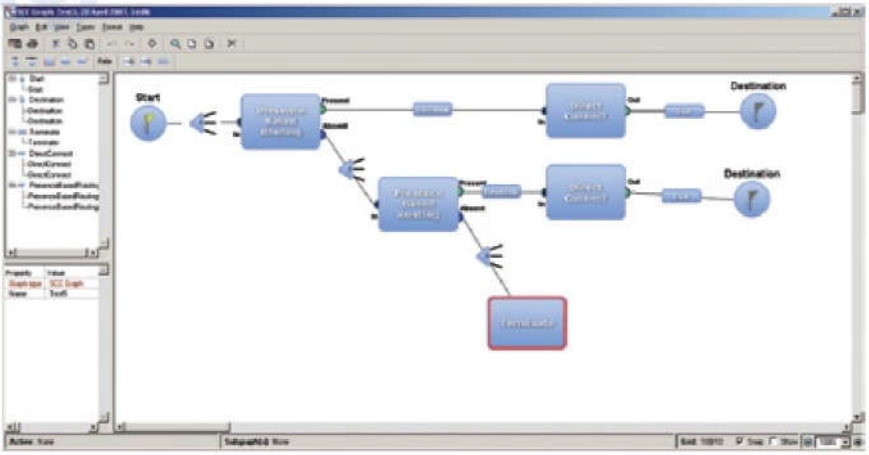

Figure 5

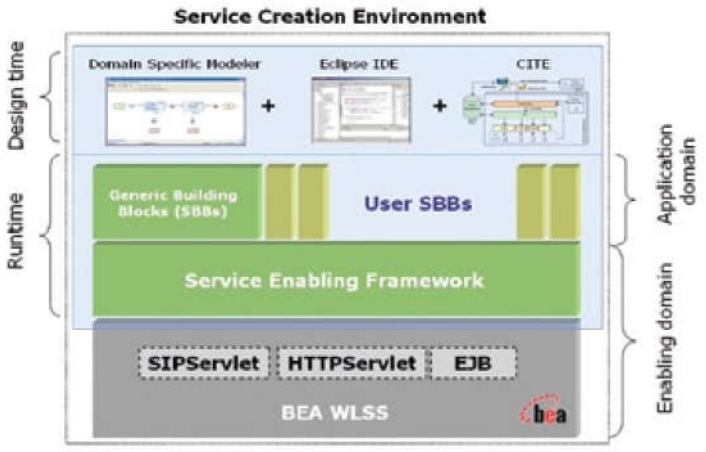

Figure 6

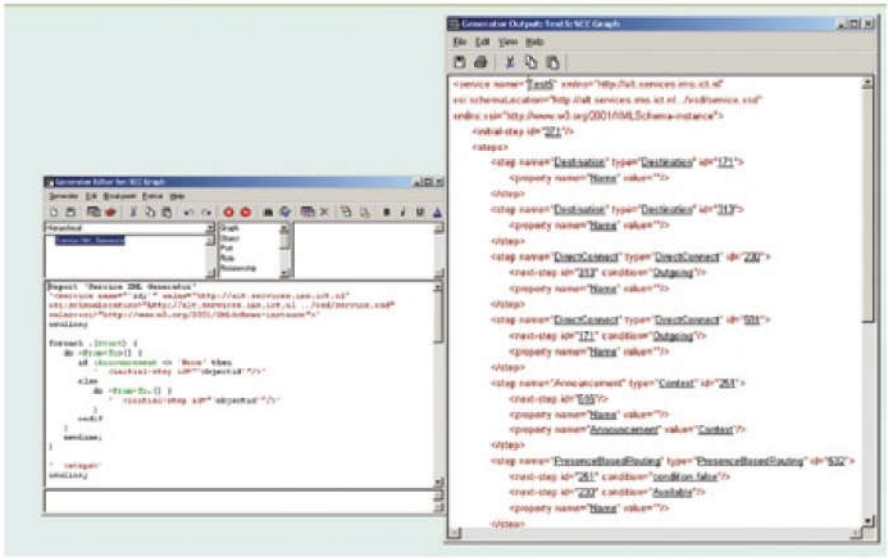

Figure 7

consequence of a higher level of abstraction. Industrial experiences, especially in areas such as automotive, military, telecom, and various other embedded devices, have demonstrated productivity increases of a factor of 3 to 10 times over traditional manual practices. For example, Nokia states that with DSM, it now develops mobile phones up to 10 times faster. Lucent-Alcatel also claims that domain-specific languages improved its productivity by 3 to 10 times, depending on the product.

In our case, we expect to achieve a significant increase in productivity over current approaches in traditional telephony and VoIP because service personnel are no longer required to handcraft and test configuration scripts. Additionally, they can reduce testing and end-user acceptance time and can discuss the service design directly with the end customer without having to explain technical details.

It is relevant to remember that these kinds of productivity gains are not new; a similar leap occurred during the transition from machine languages to 3GL. DSM follows this same recipe for success and continues to raise the level of abstraction in which developers work. General-purpose abstractions and languages, however, cannot be used to get closer to the problem domain. Instead, languages must be domain specific. A problem domain of motor control is different than that of IP telephony; therefore, the design abstractions are also different. In DSM, these domain abstractions are part of the languages used to specify the products and features.

Better alignment with the domain also offers a mechanism to improve quality. Because the rules of the problem domain can be included in the language as constraints, specifying illegal or unwanted design models can be prevented. Once experienced developers set these rules to the language, other developers follow the experts' guidelines to produce better quality designs. This is a far cheaper way to eliminate bugs – the earlier the better. The quality also improves when specifications are made using the domain terms. Specifications are then easier to read, understand, remember, validate, and communicate with, allowing customers and other stakeholders to participate in the development work.

DSM versus other solutions

Closer mapping to the business problem and automation with code generators differentiate DSM from general-purpose modeling languages. Most mainstream modeling languages such as UML focus on visualizing the code and thus fail to provide a significant improvement in productivity compared to coding in C, C++, or Java. A recent study by the Modelware project (Dubinsky 2006) that analyzed five companies using UML did not find any significant correlation between UML and increased productivity.

In our opinion, this is because of the low abstraction and modest automation possibilities. For example, UML uses direct programming concepts (classes, return values, and so on) as modeling constructs. Using 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. Furthermore, the code generated from these models requires manual completion, further reducing the already modest productivity benefits. As a consequence, developers easily find themselves making models that describe behavior and functionality, opting to write the code directly. This kills the idea of full code generation and minimizes productivity benefits.

In DSM, modeling languages are made for the purpose of enabling code generation. Instead of visualizing the programming constructs, the models visualize the problem domain. The developer creates a model of the solution in a language that contains the concepts and rules from the problem domain the developer works in. For example, when developing voice-controlled systems, concepts such as *Menu*, *Prompt*, and *Voice entry* are closer to the problem domain than assembler mnemonics. Similarly, while developing a car infotainment system, concepts like *Display*, *Channel*, and *Knob* are closer to the system than any UML concept.

Code generators in DSM read these higher-level specifications to produce the needed code. The way developers write code tends to differ from domain to domain; therefore, code generators must be specific to the given domain. While most modeling tools offer proprietary, fixed-code generators, DSM advocates the use of fully open and customizable generators for generating any type of output – not only code, but also test cases, documentation, and configuration files. The idea is for an expert to have the freedom to define these generators supported by the tools offered by DSM modeling environments. Experienced developers in the domain are best suited to write quality code in the company's problem domain.

Based on experiences with various customers, with cases that are far more complex than what is described in this article, it is clear that DSM is a preferable approach for improving software development quality and productivity in many domains. Reference cases include Nokia, ICT Solutions, EADS, Siemens, and Fuji Xerox, where DSM is used to build applications for a variety of applications such as mobile phones, definition and configuration VoIP services, and website creation for insurance products. Companies are also exploring using DSM in software development for silicon chip production machinery, nano research, and logistical systems.

As the previous example demonstrates, it is not difficult to begin using and benefiting from DSM. It is also possible to do so incrementally by first targeting only a small part of the application in the language and code generator, and then gradually adding additional parts. Finally, because the

domain changes less rapidly than the underlying technology, the approach is also future proof.

Positive results thus far

During the actual language construction, the first part of the language was constructed and then immediately tested by modeling sample applications. If the result was not satisfying (for example, allowed for the creation of illegal designs), we made the necessary corrections to the metamodel. Here tools can dramatically help. MetaEdit+ made this language testing part agile so that we could quickly test and learn what the language looked like in practice and how easy it was to create and reuse models. This minimizes the risk of making a poor language or a good language created for the wrong task. Tool support for language creation also greatly assists in finding good mappings for code generation.

The example presented in this article is part of an ongoing development project at ICT Solutions, a sister company of ICT NoviQ within the Dutch ICT Group. While the project has not yet reached its final stage of development, experiences with customers so far have been positive. The increasing interest from those involved with IMS strengthens the belief that DSM is a good approach for dealing with the speed and quality requirements of the telecommunications market in the 21st century. ECD

Angelo Hulshout *is a senior consultant at ICT Automatisering, which is in The Netherlands. He has worked for Philips Research and ICT Embedded (part of ICT Automatisering) on various topics related to software architecture for digital broadcasting systems and medical equipment. Currently, Angelo is responsible for architecture consultancy and training services at ICT NoviQ, another branch of ICT Automatisering, and is researching possible uses of DSM.*

Juha-Pekka Tolvanen *is the CEO of MetaCase based in Finland. He has been involved in model-driven approaches and tools, notably method engineering and meta-modeling, since 1991. He has acted as a worldwide consultant for method development and written 50-plus articles in software development magazines and journals. As cofounder of the DSM Forum, he plays a leading role in the shift toward model-driven development. Juha-Pekka holds a PhD in Computer Science from the University of Jyväskylä, Finland, where he also serves as an adjunct professor (docent on software development methods).*

ICT Automatisering
+31180646000
angelo.hulshout@ict.nl
www.ict.nl

MetaCase
+358144451400
jpt@metacase.com
www.metacase.com

Related readings and references

MetaCase, Nokia Mobile Phones Case Studies, 2007, www.metacase.com/papers/MetaEdit_in_Nokia.pdf

Weiss, D., Lai, C. T. R., *Software Product-Line Engineering*, Addison Wesley Longman, 1999

Dubinsky Y., Hartman, A., Keren, M., "Modelware: Industrial ROI, Assessment and Feedback," Modelware project report, 2006, www.modelware-ist.org

DSM Forum, www.dsmforum.org

