# Defining Domain-Specific Modeling Languages: Collected Experiences

Janne Luoma, Steven Kelly, Juha-Pekka Tolvanen
MetaCase
Ylistönmäentie 31
FI-40500 Jyväskylä, Finland
{janne | stevek | jpt}@metacase.com
http://www.metacase.com

**Abstract**
Domain-Specific Modeling offers a language-based approach to raise the level of abstraction in order to speed up development work and decrease the number of errors. In this paper we identify approaches that are applied for defining languages. This categorization is based on analyzing over 20 industrial cases of DSM language definition.

## 1 Introduction

Domain-Specific Modeling (DSM) can raise the level of abstraction beyond coding by specifying programs directly using domain concepts. The final products can then be generated from these high-level specifications. This automation is possible because the modeling language and generator only need to fit the requirements of one domain, often in only one company (Pohjonen & Kelly 2002, Tolvanen 2004).

This paper aims to facilitate discussion on successful DSM language creation principles. Although there exists a body of work done on language development, most of this deals only with textual languages, and concentrates on their compilers rather than the languages. In general, such research has only looked at the initial creation of the languages (e.g. Cleaveland 1988, Deursen van & Klint 1988). Fewer studies (e.g. Rossi et al. 2004, Sprinkle & Karsai 2004) have investigated the actual process of language creation, or of refinement and evolution of languages that are already in use. Moreover, the typical focus of a DSM language, providing models as input for generators, gives a special perspective to modeling language creation.

This paper identifies and categorizes approaches used for defining DSM languages. It is based on an analysis of cases that created DSM languages to automate model-based software development. Although all the DSM languages studied were implemented as metamodels and were not tied to customizing an available language, the approaches identified may also serve language creation that is based on extending available metamodels or using profiles for more lightweight language definition work.

In the next section we describe the cases and how they were analyzed in more detail. Section 3 describes the approaches identified by characterizing their main focus and by giving a representative example[1] of a DSM in that category. Sections 4 and 5 evaluate the categorization and summarize the experiences gathered.

## 2 About the studied DSM cases

This study is based on data gathered from over 20 cases of DSM creation. The cases were chosen to cover different domains and modeling: from insurance products to

[1] Due to confidentiality of industrial DSM cases, not all cases can be illustrated in detail.

microcontroller-based voice systems. Table 1 shows the cases, their problem domains and solution domains. The fourth column refers to the DSM creation approaches, which are discussed in more detail in Section 3. The cases are sorted by the fourth column for the benefit of the reader.

Table 1: DSM cases by domain and generation target

| **Case ID** | **Problem domain** | **Solution domain/ generation target** | **Creation approach(es)** |
|---|---|---|---|
| 1 | Telecom services | Configuration scripts | **1** |
| 2 | Insurance products | J2EE | **1** |
| 3 | Business processes | Rule engine language | **1** |
| 4 | Industrial automation | 3 GL | **1, (2)** |
| 5 | Platform installation | XML | **1, (2)** |
| 6 | Medical device configuration | XML | **1, (2)** |
| 7 | Machine control | 3 GL | **1, 2** |
| 8 | Call processing | CPL | **2, (1)** |
| 9 | Geographic Information System | 3 GL, propriety rule language, data structures | **2** |
| 10 | SIM card profiles | Configuration scripts and parameters | **2** |
| 11 | Phone switch services | CPL, Voice XML, 3 GL | **2, (3)** |
| 12 | eCommerce marketplaces | J2EE, XML | **2, (3)** |
| 13 | SIM card applications | 3 GL | **3** |
| 14 | Applications in microcontroller | 8-bit assembler | **3** |
| 15 | Household appliance features | 3 GL | **3** |
| 16 | Smartphone UI applications | Scripting language | **3** |
| 17 | ERP configuration | 3 GL | **3, 4** |
| 18 | ERP configuration | 3 GL | **3, 4** |
| 19 | Handheld device applications | 3 GL | **3, 4** |
| 20 | Phone UI applications | C | **4, (3)** |
| 21 | Phone UI applications | C++ | **4, (3)** |
| 22 | Phone UI applications | C | **4, (3)** |
| 23 | Phone UI applications | C++ | **4, (3)** |

All the cases applied model-based development by creating models that then formed the input for code generation. Thus, DSM language creation was not only applying modeling to get a better understanding, support communication or have documentation, but for automating development with domain-specific generators. Actually, in most of the cases the generators aim to provide full code from the modelers' perspective. This means that no changes to the generated code were expected to be needed. In all the cases, the target platform (i.e. available components and generated output language) was already chosen before the DSM language creation started. With the exception of cases that generated XML, the final detailed structure and composition of the generated output was left open and in most cases new domain framework code was created. A domain framework provides a well-defined set of services for the generated code to interface to.

Many of these domains, and hence also their respective DSM languages, can be characterized as rather stable; some however were undergoing more frequent changes. Some languages have been used now for several years whereas some have only just been created. None of the languages were rebuilt during the DSM definition process, but rather maintained by updating the available language specification. All the language definitions were also purely metamodel-based: i.e. complete freedom was available when identifying the foundation for the language. In other words, none of the cases started language definition by extending UML concepts via profiles etc. The largest DSM languages have several individual modeling languages and over 580 language constructs, whereas the smallest are based on a single modeling language and less than 50 constructs. As a comparison, UML has 286 constructs according to the same meta-metamodel as the one applied in the analyzed cases.

The data on DSM development (also know as method construction rationale, Rossi et al. 2004) was gathered from interviews and discussions, mostly with the consultants or in-house developers who created the DSM languages, but also with domain engineers and those responsible for the solution architecture and tool support. All the languages were implemented with the same tool (MetaEdit+, MetaCase 2004) and access to the language definitions (metamodels) was available for content analysis (Patton 1990) while analyzing the cases.

## 3 DSM definition approach categorization

Analysis of the metamodels revealed that the languages differed greatly with regard to their concepts, rules and underlying computational model (see samples in Figures 1, 2 and 3). The collected data indicates that the driving factor for language construct identification was based on at least four approaches:

1. Domain expert's or developer's concepts
2. Generation output
3. Look and feel of the system built
4. Variability space

This list of approaches is not complete (being based on a rather limited set of cases), nor are the approaches completely orthogonal to each other. Actually, many of the cases applied more than one construct identification approach. In the following subsections we describe these approaches in more detail and discuss how the languages' constructs were identified and defined. We also attempt to describe the process of language creation (identification, definition, validation, testing), and discuss the need for a domain framework to ease the task of code generation.

### 3.1 Domain expert's or developer's concepts

One class of DSM definitions seemed to be based on concepts applied by domain experts and developers of the models (cases 1–8 as listed in Table 1). Figure 1 shows a sample DSM of this class (case 2). All the modeling concepts are related to insurance products: an insurance expert draws models like this to define different insurance products, and then the generators produce the required insurance data and code for a J2EE website.

This type of language raises the level of abstraction far beyond programming concepts. Because of this, the generated output could easily be changed to some other implementation language. Similarly, users of these languages did not need to have a software development background, although in most cases they had. The computational models behind these languages were fairly simple, and consistent over the cases analyzed: all were based on describing static structures or various kind of flows, their conditions and order. Code was usually produced by listing each model instance separately, along with its properties and relationships to other model elements. The code generation was guided by the relationship types, e.g. code for composite structures and flow-based ordering was generated differently.

Languages based on domain experts' concepts were considered easy to define: for an expert to exist, the domain must already have established semantics. Many of the modeling concepts could be derived directly from the domain model, as could some constraints. Constraints specifically related to modeling often needed to be refined, or even created from scratch, together with the domain experts. This process was rather easy as testing of the language could easily be carried out by the domain experts themselves. If the modelers were not themselves software developers, language visualization (e.g. the visual appearance of the notation), ease of use and user-friendliness were emphasized.

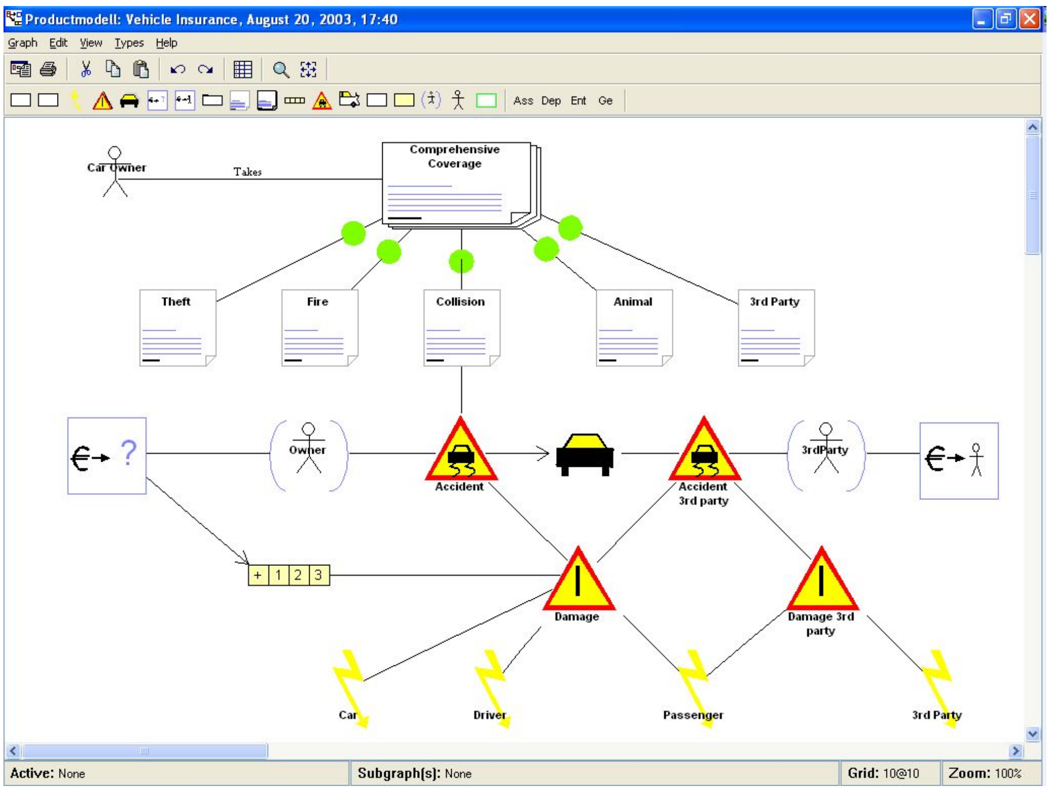

Figure 1: DSM example: modeling insurance products.

## 3.2 Generation output

One class of DSM definitions was driven by the required code structure: modeling languages concepts were derived in a straightforward way from the code constructs (cases 7–12). An example of this kind of DSM is the Call Processing Language (CPL) (Lennox et al. 2004), used to describe and control Internet telephony services (cases 8 and 11). The required XML output forms a structure and concepts for the modeling language (see Figure 2).

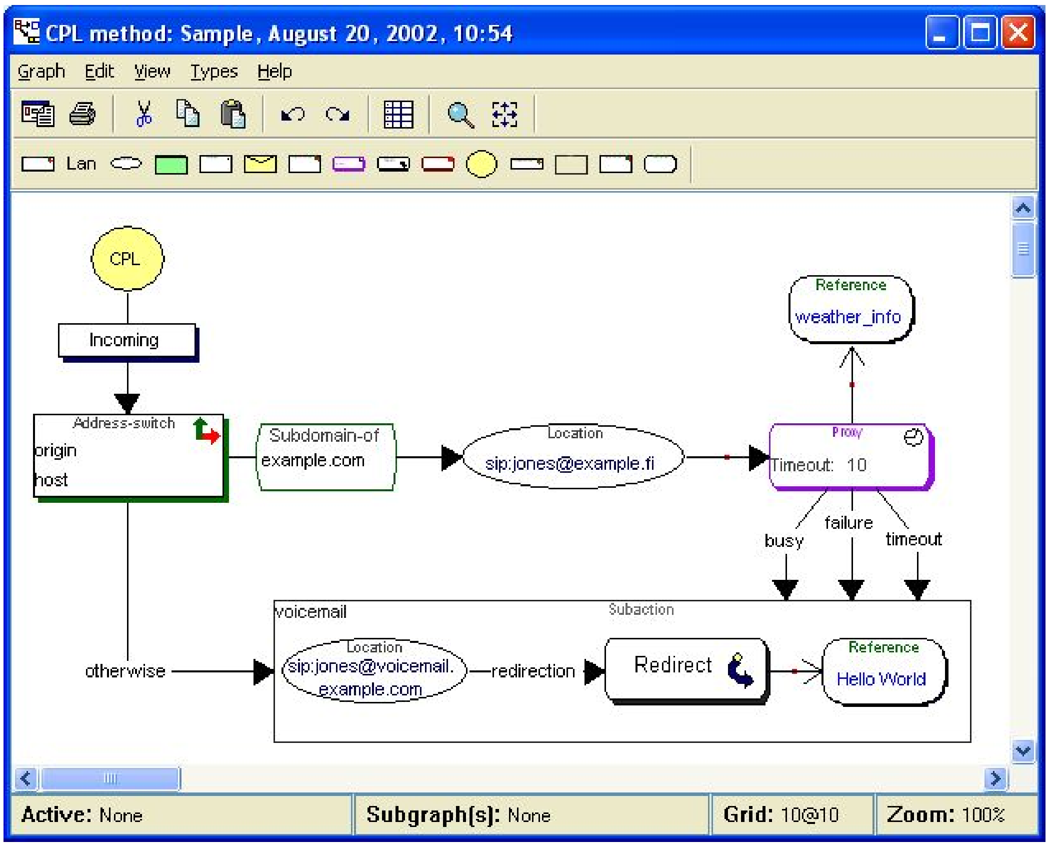

Figure 2: DSM example: Call Processing

DSM concepts to describe static parts like parameters and data structures, or the core elements and attributes in CPL and XML above, were quick and easy to define. The real difficulty was in finding appropriate concepts for modeling the behavioral parts and logic based on domain rules. This was achieved when the underlying platform provided services the models could be mapped to. This is often called analyzing the variability space (see Section 3.4). Once defined, the services and modules of the platform could even be applied directly as modeling concepts, or by having general interface concepts that allowed the modeler to choose or name the required platform service.

If a domain could not be defined or an existing architecture was not available, languages tended to use modeling only for the general static structures. The rest was done with textual specifications – often directly with programming concepts that do not provide domain-specific support.

A similar class of modeling languages are those originating from coding concepts, such as UML, schema design languages and various code visualization add-ons in IDE environments. Having models and code at substantially the same level of abstraction typically also raises the need for reverse engineering. This is similar to a class of tools, Microsoft’s Whitehorse, Rational’s XDE, Borland’s TogetherJ, that aim to offer transparency between the use of models and textual specifications.

Such a close mapping to programming concepts did not raise the level of abstraction much, and offered only minor productivity improvements. Typical benefits were better guidance for the design and early error prevention or detection. Using the CPL/XML as an example, designs could be considered valid and well-formed already at the design stage. In that way it was far more difficult to design

Internet telephone services that were erroneous or internally inconsistent: something that was all too easy in hand-written CPL/XML.

## 3.3 Look and feel of the system built

Products whose design can be understood by seeing, touching or by hearing often led to languages that applied end-user product concepts as modeling constructs (cases 11–23). Figure 3 gives an example of a language whose concepts are largely based on the widgets that Series60 and Symbian-based smartphones offer for UI application development (case 16). The behavioral logic of the application is also described mostly based on the widgets' behavior and the actions provided by the actual product. The generator produces each widget and code calling the services of the phone platform. Some framework code was created for dispatching and for multi-view management (different tabs in the pane). By using domain-specific information, much modeling work could be saved: for instance, the normal behavior of the Cancel key is to return to the previous widget. Relationships for Cancel transitions thus need not normally be drawn, but can be automatically generated; only where Cancel behaves differently need an explicit relationship be drawn.

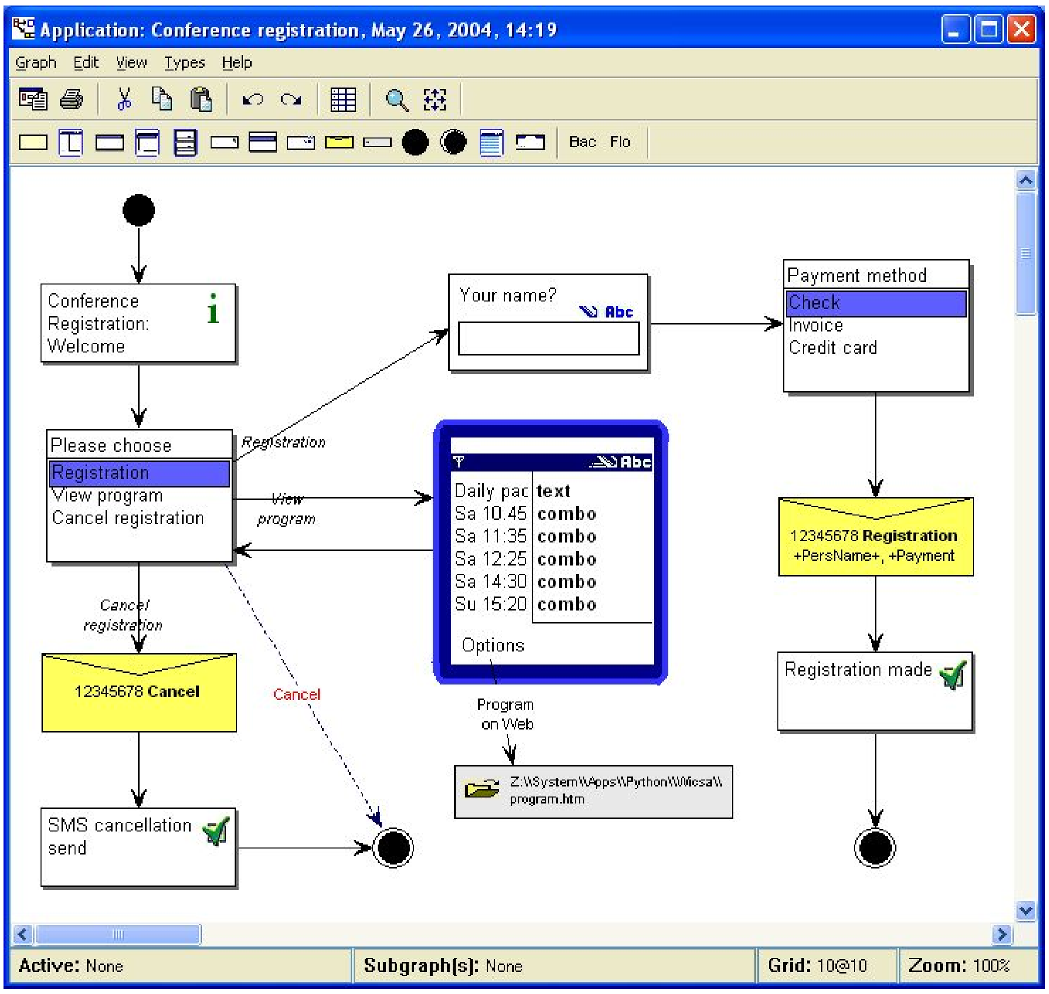

Figure 3: DSM example: Smartphone UI applications

Identification, definition and testing of the language constructs were considered easier in this approach than any other language construct identification approach. Therefore, language creation could often be carried out by external consultants with only a little help from domain experts. Although the language definition was relatively straightforward, the main challenges seemed to be in relating other types of modeling elements and constraints to those constructs originating from the look and feel. If the look and feel constructs were sufficiently rich to also cover functionality, the level of abstraction of modeling was raised substantially beyond programming.

In many cases, the look and feel based cases had an existing framework, product platform or API, which formed a reasonably solid foundation for the key modeling language concepts. The APIs varied in their levels, from very low-level APIs near the code, to very abstract operations and commands. The simpler generators usually produced the code as a function per widget or similar state, with the end of the function calling the next function based on user input. Tail recursion was used to reduce stack depth where necessary. More complex generators produced state-based code, either in-line or as state transition tables. None of the languages based on look and feel required frequent reverse engineering, but some called for importing libraries as model elements. Usually only interfaces were required for these libraries, but in at least one case components with their implementation (i.e. whitebox) were needed. Generators targeting other implementation languages were not defined, although that was considered possible to achieve.

## 3.4 Setting the variability space

The final language definition approach was based on expressing variability (cases 17–23). Such cases were typical in product families, where languages were applied for variant design. Typically, the variability space was captured in the language concepts, and the modelers' role was to concentrate on the issues which differ between the products. All the cases that were based on describing variability had a platform that provided the common services the generated code interfaced with. This interfacing was typically based on calling the services of the platform, but there were also cases where generators produced the component code.

Languages describing variability were among the most difficult DSMs to create. The main reason was the difficulty to predict the future variants. This called for flexible language definitions that were possible to extend once new kinds of variations arose. Languages for pure static variability (often for configuration) were found relatively easy to create, however. The difficulty lay in behavioral variability and coming up with a language that supported building almost any new feature based on the common services of the platform. The success of the language creation was dependent on the product expert's knowledge, vision to predict the future, and insight to lay down a common product architecture. Therefore, the role of external consultants to support DSM creation was often smaller than with other approaches. In the best cases, though, the external consultant's experience of DSMs and generators complemented the expert's experience in the domain and its code. This normally required a consultant who was himself an experienced software developer (although not in that domain), and an expert who was not too bound to a low-level view of code.

In these cases language constructs were explored using domain analysis to identify commonalities and variabilities among the products to be built using model-based code generators. For example, Weiss and Lai (1999) present a method to detect commonality and variability of both static and dynamic nature. Each variation point will be specified with variation parameters. By setting parameters for variation it offers a clear starting point for language concepts, like proposing data types and their variation space as well as constraints for combining variability. Feature modeling (Kyo et al. 1990) was not applied to explore variability as it was found to operate at a level too general to identify DSM concepts. Feature models do not capture the dependencies and constraints that are required to define modeling constructs. Among the studied cases, product architecture served better to find product concepts and their interrelations.

A product family platform and its supporting framework also have a notable influence on the modeling language concepts and constraints. Commonalities were usually hidden into the generator or framework in addition to complex issues which can be solved in an automated generator. In many cases there were several different computational models used to support all the required views of the systems. For example, in embedded product families, it was common to follow the state machines with domain specific extensions to best describe the system's behavior and interactions.

The level to which abstraction was raised was dependent on the nature of variability. As would be expected, cases where the variability could be predicted reasonably well showed higher levels of abstraction than those where future variability could not be pinned down. A common solution for these latter cases was to make the modeling language and generators easy to extend, allowing the level of abstraction to be raised substantially now, and making it possible to maintain that level in the future.

## 4 Evaluation of the Categorization

After having categorized the cases according to which of the four approaches were used, we noticed that each case had used only one or two approaches. Further, where there were two approaches, only certain pairs of approaches seemed to occur. Of all 16 possible pairs made up of a primary approach and a secondary approach, only 5 were actually found in the data. This prompted us to re-order the categories into the order now shown (previously generation output was last), so that each case used one approach and its successor or predecessor.

Cases performed mainly by the customer mostly occur early in the list. Conversely, those cases which had been performed by more experienced DSM practitioners tended to come later in the list. The order of approaches thus probably reflects an order of increasing DSM maturity.

Some cases were found to resemble others from the language point of view, although the product domain and generated code were different (e.g. the cases of ERP configuration and eCommerce marketplace).

Approach 1, domain expert's concepts, seems to provide little insight. In some cases it simply means that somebody else identified the concepts, and we thus lack the information of which of the other approaches they used. In the three cases where the customer was not mainly responsible for the concept identification, the DSM project has not progressed beyond an initial proof of concept. These cases thus probably reflect domains that are immature, and where the DSM consultants lacked previous experience that would have enabled them to raise the maturity in that domain.

In approach 2, generation output, there were significant differences between those cases whose generation output was itself an established domain-specific language, and those where the output was a generic language or an ad hoc or format such as a configuration file. Those cases worked best where the output was an established domain-specific language, because the domain was more mature and the company in question already had a mature implementation framework, either their own or from a third party. In both CPL cases, the companies wanted their own additions to the languages, further improving the domain specificity.

When the output is in a generic programming language, it would often be better apply an approach other than generation output, to truly raise the level of abstraction. When the output is to an immature format, it would often be better to analyze the domain further to improve its understanding and the output format, rather than build a direct mapping to the existing shaky foundation.

Approach 3, look and feel, can be regarded as the first of the four approaches that consistently yields true DSM solutions. It is thus a valid approach to apply in new DSM projects, whenever the end product makes it possible. It was also the most commonly applied approach, found in 13 out of 23 cases.

Approach 4, variability space, was only found in combination with approach 3. The cases where it was the primary approach, 20–23, were all in the domain of phone UI applications, generating C or C++ (case 16 was a simpler domain, a subset of these). These cases are certainly among the most complex, and this partly accounts for the similar solutions. A second major factor is that experience with previous similar cases had provided a proven kind of solution for this domain. Whilst each language was created from scratch, the knowledge of previous cases from this domain certainly influenced the way the cases were approached. The resulting DSM languages and in particular generators differed substantially, reflecting the different needs of the domains, customers and frameworks.

The use of the variability space approach in the radically different domain of ERP configuration (17 & 18) shows that this approach is not restricted to state-based embedded UIs. Perhaps the most likely explanation for this clustering of cases is that this approach requires the most experience from the language creators, and yet also offers the most power. In particular, the combination of the almost naïve end-user view of the look and feel approach with the deep internal understanding of the domain required by the variability space approach seems to yield the best solutions, particularly in the most complex cases. When used together, the look and feel approach tended to identify the basic concepts, and the variability space approach helped define relationships and what properties or attributes each concept should have.

## 5 Conclusion

In this paper we have discussed approaches to identifying concepts for DSM languages, based on experiences collected from over 20 real-world cases. The cases show that there is no single way to build DSM languages: more than one language creation approach was applied in the majority of cases. In the cases studied, we identified four different approaches used by the domain expert, expert developer or DSM consultant.

Of the four approaches in our categorization, the first relied on the domain expert’s intuition or previous analysis to identify concepts. This approach is essential in that it emphasizes the role of the expert, but forms a weak point of the categorization in that the experts themselves must normally have applied one of the other approaches. The second approach identifies concepts from the required generation output, and can only be recommended where that output is already a domain-specific language. The third and fourth approaches, end product look and feel and variability space, seem to be the best overall, although not applicable in every case. Using them together was particularly effective.

Defining a language for development work is often claimed to be a difficult task: this may certainly be true if you want to build a language for everyone. The task seems to become considerably easier when the language need only work for one problem domain in one company. According to the cases analyzed the main difficulties are found in behavioral aspects and in predicting future variability. Almost all cases with both these difficulties required experienced DSM consultants, and all used more than one approach to identify concepts.

In all cases, DSM had a clear productivity influence due to its higher level of abstraction: it required less modeling work, which could often be carried out by personnel with little or no programming experience. The increase in productivity is not surprising, considering that research shows the best programmers consistently outperform average programmers by up to an order of magnitude. DSM embeds the domain knowledge and code skill of the expert developer into a tool, enabling all developers to achieve higher levels of productivity.

This paper aims to facilitate discussions on DSM by summarizing and analyzing our experiences of how DSM language creators identify and define modeling constructs. More research work is needed to better understand the DSM creation process, and to disseminate the skills to a wider audience. Particularly welcome would be empirical studies that cover more cases from various domains, and using different metamodeling facilities. As DSM use grows, research methods other than field and case studies would also be welcome, for example surveys and experiments.

## References

Cleaveland, J. C., (1988) Building application generators, *IEEE Software*, July.

Deursen van, A., Klint, P., (1988) Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75-92.

Kyo, C., K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, A.S. Peterson, (1990) Feature - Oriented Domain Analysis (FODA) Feasibility Study, Technical report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University.

Lennox, J., et al., (2004) CPL: A Language for User Control of Internet Telephony Services. Internet Engineering Task Force, IPTEL WG, April.

MetaCase, (2004) MetaEdit+ Method Workbench 4.0 User's Guide, www.metacase.com

Nokia (2003) Series 60 SDK documentation, version 2.0, 2 (www.forum.nokia.com/)

Patton, M., (1990) Qualitative Evaluation and Research Methods, Newbury Park, Sage, 2nd edition.

Rossi, M., Ramesh, B., Lyytinen, K., Tolvanen, J.-P., (2004) Documenting Decisions in Method Engineering by Method Rationale, *Journal of AIS*, accepted, September 2004 (to appear).

Pohjonen, R., Kelly, S., (2002) Domain-Specific Modeling, *Dr. Dobb's Journal*, August.

Sprinkle, J., Karsai, G., (2004) A domain-specific visual language for domain model evolution, Journal of Visual Languages and Computing, Vol 15 (3-4), Elsevier.

Tolvanen, J.-P., (2004) Domain-Specific Modeling (in German: domänenspezifische Modellierung) *ObjektSpektrum*, 4, July/August.

D. Weiss, C.T.R. Lai, (1999) Software Product-line Engineering, Addison Wesley Longman.