Integrating Models with Domain-Specific Modeling Languages

Juha-Pekka Tolvanen
MetaCase
Ylistönmäentie 31
FI-40500 Jyväskylä, Finland
+358 14 641 000
jpt@metacase.com

Steven Kelly
MetaCase
Ylistönmäentie 31
FI-40500 Jyväskylä, Finland
+358 14 641 000
stevek@metacase.com

ABSTRACT

Model integration is inescapable: any non-trivial system will be too large to fit sensibly in a single model. The model will have to be split, maybe into different aspects or languages, different modeler roles and tasks, different phases of the software development life cycle, etc. In Domain-Specific Modeling, the possibilities to integrate models are fundamentally better than with general-purpose languages as the company has full access to the language definitions. We describe and compare different ways to integrate DSM models, based on real world experience of what has been shown to work in practice on industrial scales.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**] Design Tools and Techniques - *user interfaces, state diagrams* D.2.6 [**Software Engineering**] Programming Environments - *programmer workbench, graphical environments* D.3.2 [**Programming Languages**] Language Classifications - *Specialized application languages, very high-level languages*

General Terms

Design, Languages.

Keywords

Domain-specific modeling, language integration, metamodel

1. INTRODUCTION

To model any non-trivial software system inevitably requires the integration of multiple models (‘model’ here means a single graph, often represented as a diagram.). A large domain further requires the use of multiple modeling languages and representational views. Splitting the system over different languages and models can be done in various ways: modularization of the system; different aspects of the system; different people and their roles in development; different phases of the development life-cycle. In all cases there is a need and expectation that individual models and their elements can be linked and integrated with other models. This reintegrates the separate models into a cohesive whole, allows modelers to see and reuse work done by others, and allows the checking of system-wide properties.

This paper examines approaches for integrating modeling languages. We focus solely on Domain-Specific Modeling (DSM) languages, as opposed to General-Purpose Languages (GPLs). This difference is important since with DSM the companies have full control of the individual languages and how they can be integrated. Since there is plenty of work focusing on the technical side of integration (such as model transformation languages and metamodeling languages), we focus on integration needs and approaches coming from the higher, tool-independent level of language engineer work in practice. Where possible we refer to industrial experiences from public cases including Porsche [1], Polar Electro [2] and Panasonic [7].

We start by introducing an example domain for model integration: embedded UI applications. We use this to compare and contrast the two basic integration paradigms, string matching vs. direct reference. We then identify and describe different ways to integrate models, first looking at those that can be accomplished independently of the modeling language, e.g. to allow modelers with different roles to use the same model in their own ways, and then at ways of integrating the modeling languages themselves to provide the best integration on the model level.

2. INTEGRATION EXAMPLE

To demonstrate the different integration alternatives we use a common domain in the article. We show how domain-specific languages can be integrated for developing embedded user interface application. This is a commonly addressed domain with DSM and public solutions are presented in automotive [1], home automation [7], medical devices [2] and mobile phones and professional radios [4]. It is useful for our purposes as it spans multiple people, roles and phases of the development life cycle, allowing us to consider the whole range of factors that may require multiple languages and views.

Figure 1 shows a model of a sample UI application, for making shopping lists on a mobile phone. The model shows the use of the various UI widgets, the navigation between them, and access to phone services.

Developing UI applications covers many tasks, all of which may involve modeling: concept demonstration, prototyping, interaction design, localization, implementation, testing, etc. Often these tasks are performed by different persons and some tasks occur in parallel. For example, while interaction designers are still seeking optimal usability, localization and application implementation may have already started. The same interaction design information is being read and updated at the same time. This calls for highly integrated languages and views to the models, as well as good tool support for multiple simultaneous modelers.

In some industries the specification and realization steps occur in different companies. For example in automotive [1] it is common that car manufacturers make specifications and subcontractors provide applications along with hardware. In other industries, like home automation [7], the developers generally work within the same company.

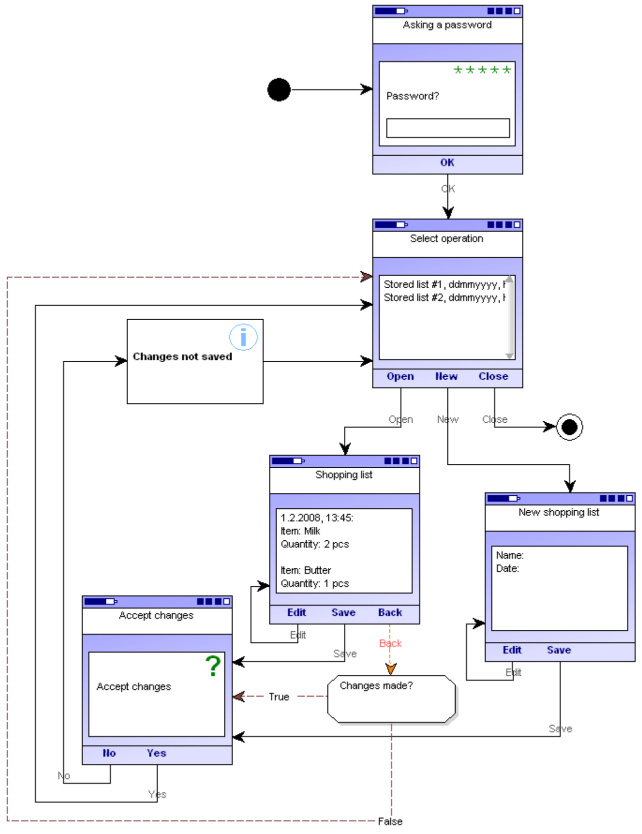


Figure 1. A sample model for interaction design.

3. INTEGRATION PARADIGMS

Several approaches have been suggested as the underlying mechanism to support model integration. We shall ignore simple duplication – the deep copying of whole elements to multiple models – because of its obvious problems, as seen for example in the PIMs and PSMs of original UML-based MDA and its model-to-model transformations [6]. The remaining approaches fall into two main paradigms: string matching and direct object references.

3.1 Splitting and string matching

Simplistic modeling tools provide no support for integration: you can save a model as a file, often as XML, and often only containing a single diagram. Elements within that model can refer to each other, but not to elements in other models (i.e. other XML files).

Such tools inevitably face the need for some mechanism to allow references outside a single diagram. Typically, the early approach is based on simple filenames and string matching: e.g. look in “Shopping List.xml” for the element called “New shopping list”. Variants of this approach include omitting the filename and searching through all files in the current directory, path, project etc.; specifying the type of the model element to search for; and using XML ids rather than name properties.

Simplistic modeling tools generally start this way, splitting models into small, separate files and recombining them based on string matching. Multiple source files and string-based references are of course familiar from textual programming, which may make adoption easier.

Textual DSLs will of course also follow this paradigm; indeed, in text even links within a single file are accomplished by string matching, as seen in Listing 1.

Listing 1. Textual DSL, "Shopping list" string matches in bold

```
WIDGET Select operation
        ON Open GOTO **Shopping list**
WIDGET **Shopping list**
        ON Save GOTO Accept changes
        ON Edit GOTO **Shopping list**
        ON Back GOTO Changes made?
```

Pseudo-textual tools, e.g. JetBrains’ MPS, actually fit into the other paradigm, which we shall look at next.

3.2 Direct object references in a repository

As we have seen, textual languages use string matching even within a single file; XML models can use direct references within a single model file but string matching outside. The next logical step is to expand the space in which direct references are possible, to encompass all the models in a project.

This is generally talked of as the repository approach, as used for example in MetaEdit+ [5]. The models and their elements can be considered like the objects in a running object-oriented program: they all have their own independent existence, and can refer to each other directly. This of course also nicely matches the graph structures visible in model diagrams, at least to a first approximation.

Of course, where desired the references can still be made by string matching, i.e. one object including the explicit name of another object, e.g. to add a level of indirection. With both options available, language engineers can look for the most appropriate model integration approach for their situation.

3.3 Choosing an integration approach

Both integration approaches have their place and both approaches should be available for language engineers to consider. While different options are possible the suitable language integration approach should be investigated from the modeling process point of view:

- Who is going to use the models - create, read, modify? What are the different persons and roles involved?
- How is the work of different developers? Are there certain preferable ways for reusing others work?
- When would it be best, or possible, to integrate models from different developers?
- How do changes in the life-cycle influence to the models already made?
- Performance considerations: Speed of network, distance of developers, speed of the tooling; size of models.

- Versioning and integration into existing processes.

There can also be business reasons for keeping the models separate: A company may not want to reveal everything to its subcontractors, or may want its customers to only use certain DSLs to modify part of the system.

4. INTEGRATING MULTIPLE ROLES IN A SINGLE LANGUAGE

In the following we describe some of the main alternatives to integrate different roles or tasks of users, without having to have a different model for each role or task. If we can make a single model usable for two distinct user roles, without having to split the model in two, we reduce the friction caused by splitting and reintegration.

4.1 Different representations

Some tools offer support for multiple views, filters or representations of the same model, with no extra work required of the language engineer creating the metamodel. For instance, the same set of model elements could be represented in two different diagrams, e.g. one showing navigation flow (Figure 1) and another showing data use (Figure 2). Since the shared model elements are the same among both diagrams there is no extra work needed to keep models up-to-date and consistent with each other. For example, a change of name from "Asking a password" to "Request password" will be seen in both representations.

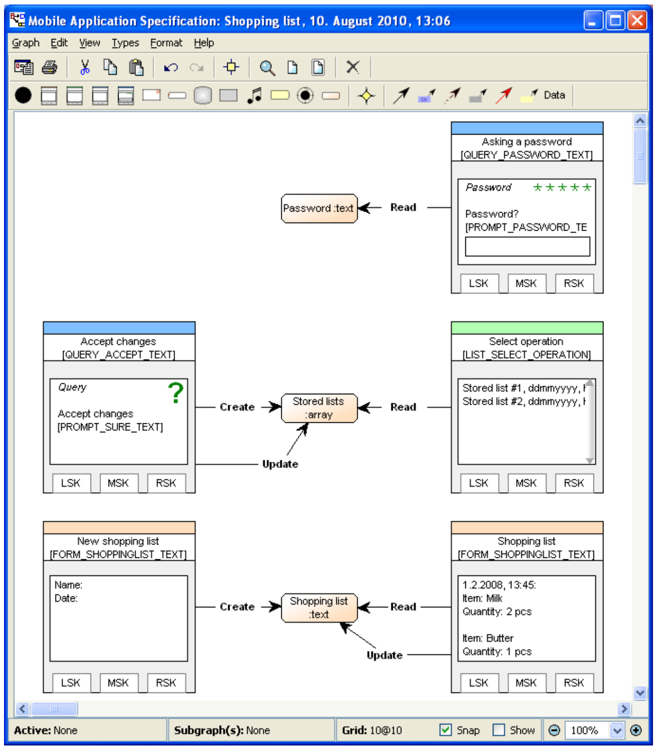
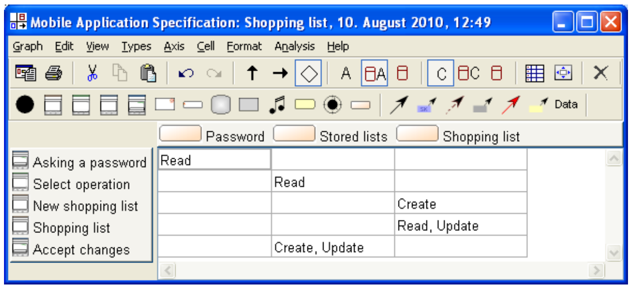


Figure 2. A diagram for Figure 1's model showing data use.

Figure 1 could alternatively be complemented by a matrix representation, as in Figure 3, which gives a better overview of which elements access which data elements. As before, each model element exists only once, just with multiple representations of the shared elements.

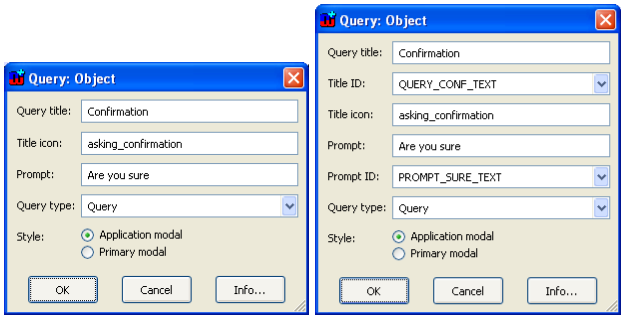

Mobile Application Specification: Shopping list, 10. August 2010, 12:49
Graph Edit View Types Axis Cell Format Analysis Help
Data

	Password	Stored lists	Shopping list
Asking a password	Read		
Select operation		Read	
New shopping list			Create
Shopping list			Read, Update
Accept changes		Create, Update	

Figure 3. A matrix representation.

4.2 Different tool behavior

One approach is just to hide part of the model data in the tool user interface, and/or prevent the user from entering or editing it. For example, in MetaEdit+ [5] the dialogs used to edit and view model data can be modified to be suitable for different users. Tool behavior is thus changed for different needs rather than creating different versions of the language. In Figure 4, the dialogs show two different views on the same underlying model. The dialog on the left shows the data needed for interaction designers and the dialog on the right information about implementation details.



Query title:	Confirmation
Title icon:	asking_confirmation
Prompt:	Are you sure
Query type:	Query
Style:	[✓ Application modal] Primary modal

Query title:	Confirmation
Title ID:	QUERY_CONF_TEXT
Title icon:	asking_confirmation
Prompt:	Are you sure
Prompt ID:	PROMPT_SURE_TEXT
Query type:	Query
Style:	[✓ Application modal] Primary modal

Figure 4. Property dialog for interaction design (left) and for detailed view (right) on one modeling concept.

If some development role requires access to only certain kinds of model data a viable approach may be to extend this beyond dialogs to give subsets of the whole language, often by hiding the rest. For example, technical engineers may want to see all the data whereas for localization it is enough to see only the elements that will appear directly to the end user.

While both these examples had one view as a subset of the other, that is not always the case: both views could be subsets of a larger, complete, set of information, with no one view showing everything. In practice, however, this seems rare.

Creating such alternative tool behavior has a cost, so will probably not be done if the differences between the needs of users are small: users simply see the whole model and ignore the parts that are not relevant to their role.

4.3 Different notations

A more advanced approach is to provide alternative visualizations of the same model data: The language definition may provide different notations to different users of the models. For example, in [7] a view closer to a realistic, pixel accurate, notation was required. While in some cases close imitation of final products greatly improves readability and validation of models, often a more abstract notation allows seeing important details that would be swamped by the realistic view.

A proven approach is to define different sets of symbols for the language concepts. These different symbol sets can then be selected in different stages and by different persons. Figure 5 shows the same model data as in Figure 1, but now from the angle of detailed design. The choice of which symbol set to use can be made globally, for a certain diagram, or per element; in this case the notation was selectable for each diagram.

Tools that support multiple simultaneous modelers can use this approach to enable collaboration and early feedback during modeling work. Different visualizations can also be applied to show errors, inconsistencies or incompleteness of the model. The model checking can be visualized for example by using special coloring, icons or special texture. In Figure 5, red text is used to illustrate errors and missing data. For example, the red text "Undefined!" is shown for the "Asking a password" dialog to indicate that its Title ID property has not yet been filled in.

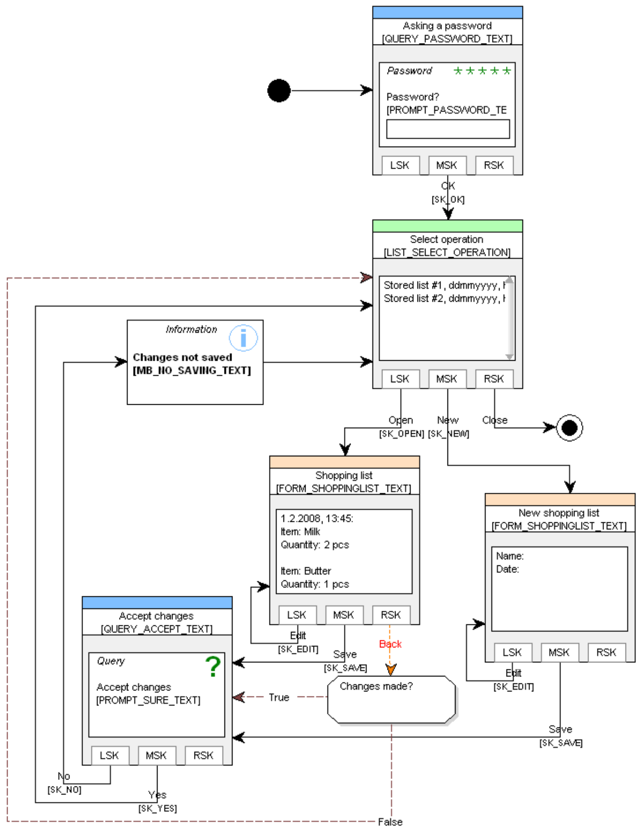
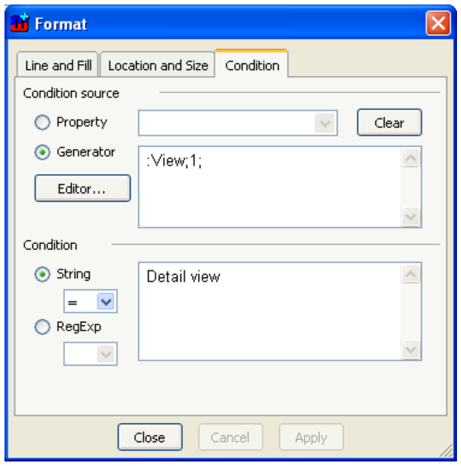


Figure 5. Detailed view of the same application model diagram as illustrated in Figure 1.

Technically the different notations can be defined by specifying them all as conditional parts of a single symbol for the concept, or by importing them from external files when the view must be changed. The former is particularly good if the amount of alternative choices is small and the notational elements can be defined in advance. Including the notation to the language definition also simplifies sharing and using the language among other modelers. Figure 6 shows how a symbol element is made conditional based on the choice of realistic or detailed view.



Condition source

Property

[✓ Generator] :View;1;

Condition

[✓ String] = Detail view

RegExp

Figure 6. Defining conditional symbols for a language.

The latter, importing notation from external sources, is preferable if the number of alternative visualizations per language element is big and the individual notational elements are not yet known. Separating notational elements from the rest of the language definition does however make its management, sharing, versioning, and updating more challenging.

Notation elements may also be partly derivative: a symbol may have parts that fetch information from other models. The symbol defines the path to the information either declaratively or as a script or generator. This is particularly useful when data is needed in read-only mode: model data is available but it is not allowed to be changed from that point of the language. For example, some of the details specified in a submodel, such as ports, are shown in the upper diagram to give the wider context.

5. INTEGRATING LANGUAGES

A more powerful approach for integrating models is to integrate the languages, their underlying concepts and constraints. The following approaches have been identified to be used in practice.

5.1 The best integration is no integration

In GPLs, it is often impossible to integrate several points of view or aspects in a single language (i.e. diagram type) without that language and its models becoming too big. In DSM, the modeling language for a given aspect in a given domain would be smaller than for that aspect in all situations. It thus becomes possible to fit several aspects into one language without it becoming unwieldy. This also has the benefit that a single diagram can express several aspects in one coherent view, rather than the modeler having to split the information over several diagrams, maintain the links, and reconstitute them together mentally to see the whole picture.

5.2 Relationships between models

The simplest integration is where an element in one model points to another model, often to describe the internal details of that element. For example, in Figure 5, the details of the Shopping list element, like list content, could be described in another (sub)model. It is the task of the language engineer to define what kind of relationships could be made between models, such as:

- If a model element can have more than one (sub)model
- If several model elements can have the same (sub)model

- If the submodel is the same language as the parent model or a different language.

This kind of relationship between models can be used for various purposes. When several model elements can refer to same submodel, the top level model elements can be used to configure the submodels: each model element then adds its own details to the common part.

With relationships to several kind of models (each having a different language) different aspects can be separated to their own models. For example, the shopping list could be described in more detail from the content and operation point of view – and for describing both views there are different (sub)languages.

Often the relationships among models are not just targeting another whole model but refer to the content of the other model. This calls for sharing the language concepts among the languages.

5.3 Common language concepts

Where the model information is split over several diagrams, it can also be useful to allow reuse of elements between models, e.g. to provide different perspectives on the same model element. The same language concept will then be used in multiple languages.

For example, in [1] a set of integrated languages for developing automotive infotainment systems is presented (see Figure 7). The approach identifies a set of roles such as graphical designers defining the layout elements (fonts, icons, colors etc.), usability experts defining the structure and layout of individual displays and developers specifying the interaction logic and behavior. To support model integration among the various persons and their roles the languages are defined into a common metamodel of several sublanguages, with shared domain concepts.

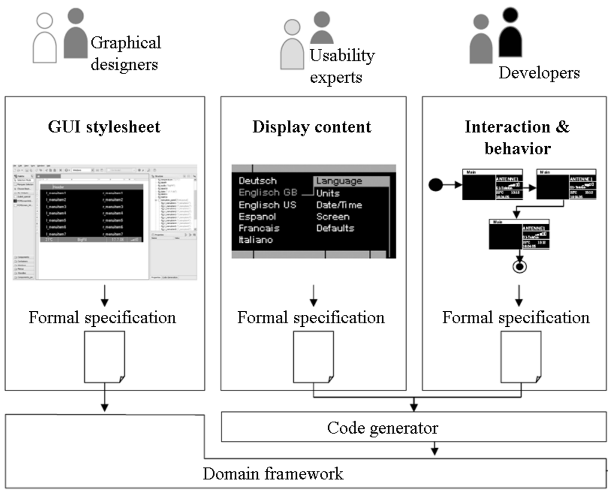


Figure 7. Different languages for different roles (from [1]).

Sometimes elements are reused by placing them directly in several diagrams; other times an element may be directly in one diagram, but included in another diagram only as a reference or property of an element there. For example, in Figure 7 each state in the right-hand column refers to a Display defined in the middle column.

On the metamodel level this means that Display object is shared among the different modeling languages. Figure 8 shows the metamodel for this (simplified for this example). The language definition on the left hand side, called Display content, is used to define individual displays and their detailed structure. The language on the right hand side, called Interaction & behavior, has a State object which has a reference to one Display definition. The reference can be seen in Figure 8 (highlighted here by the red dashed arrow). This language integration structure allows developers to refer to existing display definitions while focusing on interaction design.

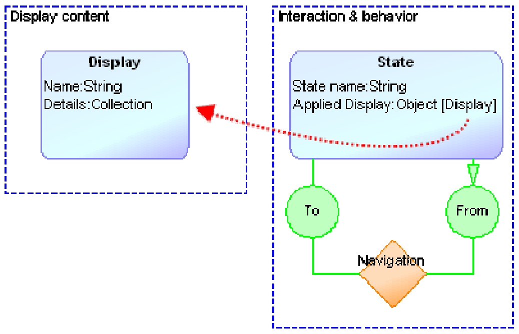


Figure 8. Using the same language construct, Display, in two different languages.

Integrated language definition can also determine the workflow: how individual model elements should be reused. An individual display defined by usability experts can be reused in the models describing interaction design. Interaction designers and technical developers can then reuse the defined displays and decide if they want changes made to the reused displays to be applied in interaction designs.

The integrated metamodel may give different names or labels for the shared concepts. Usability experts may want to call a thing "display" while technical developers speak about a "state" having a "view".

A language may also include constraints to define if reuse is mandatory, constraining where elements can be reused from, and who can create new reusable elements. In the case of the display concept, the language engineer could define that interaction designers may only reuse existing displays, or that they could define and use their own displays but that these would only be local, not available for reuse by other designers.

5.4 Creating a metamodel from models

A special case of model integration is when a first model M_1 effectively defines a language to be used for writing a second model M_2. M_2 is thus an instance of M_1, and M_1 therefore determines the very form of the data that can be entered into M_2, not just some constraints on the content of that data. Examples of this can be found from generic languages, e.g. UML's Class Diagram and Object Diagram. For each attribute defined in a class in a Class Diagram M_1, the corresponding object in the Object Diagram M_2 should be able to provide a value for that attribute – but not change the details of the attribute such as its name or type. Similarly, in a language workbench the language used to define metamodels allows the metamodeler to create an M_1, and a modeler can then create an M_2 (or several) that are instances of M_1.

This approach can also be used in cases where there is a strong dependency that what is legal in M_2 is determined by the contents of M_1, even though it may not initially be seen as a clear-cut case of instantiation. The primary mechanism offered by a language workbench for constraining models is the modeling language, so leveraging this mechanism to handle the constraint enforcement may be better than trying to cobble together an ad hoc solution.

In industrial use of MetaEdit+ we have seen several cases where the customer initially wanted strong constraints from one model to another. On closer examination the models were to be made by different people; those building the second models should have only read-only access to the first model; and the process for defining the first model would be stricter. Often the first model was talked about as "defining the components that the second models would use". Sometimes there would be one first model and several second models, or then each second model could "import" a set of first models, which would together provide the set of legal components to use.

Allowing the first model (or a set of them) to form a metamodel for the second models fit these requirements well. In most cases there was a base metamodel for all the second models, which was then extended in parts by the contents of the first models. The automation, integration and evolution facilities of MetaEdit+ allowed the modelers to create a second model then add references to extra first models, and have the modeling language be extended on the fly to include the new concepts.

Technically, it would have been possible to have the builders of the first model use the normal metamodeling language or form-based metamodeling tools of MetaEdit+. Instead, we used the natural form of the language for the first domain, and simply made generators for that language that could create the desired corresponding metamodel for use by the second models. With the appropriate generator, any model in MetaEdit+ can thus effectively be a metamodel.

6. RELATED WORK

The splitting and string matching of 3.1 can also be a deliberate policy, as in [9]. The wider questions of string matching vs. direct references and model versioning are covered in more detail in [3].

To our knowledge, out of the box no other tool offers multiple simultaneously editable representational paradigms of the same model, as in 4.1. Extending a model as in 4.2 is accomplished in [8] by storing the extra information in external XML files.

A similar approach to the common language concepts of 5.3 is the use of "gateway metaclasses" in [8]. However, in that case whole elements are reused by copying, not by reference; to avoid that duplication it is suggested to have one end of the reference just use the element's name, with matching based on identical names. The ModelBus add-on to Microsoft DSL Tools has a similar approach.

7. CONCLUSIONS

In DSM, language engineers have full control over the languages, and can thus decide on an appropriate model integration approach for their situation. Sometimes it can be possible to integrate several areas of interests, e.g. persistency, navigation, layout, data, into a single modeling language, whereas at other times the use of different languages, and explicit integration among the models is preferred. In any case, there will always be multiple model diagrams to integrate, whether by elements having subdiagrams, or elements being reused or referenced in several diagrams.

We have described some of the main model integration approaches by analyzing language integration cases based on our consulting work in various industries, including automotive, telecom, and consumer electronics. The approaches described have been illustrated by extending a common base example model.

The range of integration approaches described probably reflects our experience with a repository-based tool, which makes linking and reusing across models and languages easy. While all of the approaches could in theory also be implemented in a tool based on separate files, e.g. XML files with string matching, the amount of work necessary to provide good support to modelers with those tools may be prohibitive in some cases.

Whatever the tool, avoiding the need to split and recombine data is a useful tactic where possible. A single language integrating multiple aspects, tool support that can present different views for different users, and multiple representations and notations of a single language are all approaches that can achieve this.

Where it is necessary or desirable to split information over multiple models, possibly of different languages, they can be kept integrated by shared elements, relationships between models, and even by creating metamodels from models on the fly.

It is important that there are several possibilities for integration and that tools applied do not limit the language engineer's, and later modelers', choices on how best to integrate models.

8. REFERENCES

[1] Bock, C., Görlich, D., and Zühlke, D. 2006. Using Domain-Specific Languages in the Design of HMIs: Experiences and Lessons Learned, In *Proceedings of the MoDELS'06 Workshop on Model Driven Development of Advanced User Interfaces*, CEUR Workshop Proceedings, Vol-214.

[2] Kärnä, J., Tolvanen, J.-P., and Kelly, S. 2009. Evaluating the Use of Domain-Specific Modeling in Practice, In *Proceedings of the 9th OOPSLA workshop on Domain-Specific Modeling*. http://www.dsmforum.org/events/DSM09/Papers/Karna.pdf

[3] Kelly, S. 2010. Mature Model Management, *ObjektSpektrum*, October 2010 (in German, to appear).

[4] Kelly, S., and Tolvanen, J-P. 2008. *Domain-Specific Modeling: Enabling Full Code Generation*, Wiley.

[5] MetaCase 2008. *MetaEdit+ Workbench 4.5 SR1 User's Guide*, http://www.metacase.com/support/45/manuals/

[6] OMG 2003. *MDA Guide V1.0.1*, http://www.omg.org/cgi-bin/doc?omg/03-06-01

[7] Safa, L. 2007. The Making Of User-Interface Designer, A Proprietary DSM Tool. In *Procs of 7th OOPSLA Workshop on Domain-Specific Modeling*, Technical Reports, TR-38, University of Jyväskylä, Finland. http://www.dsmforum.org/events/DSM07/papers/safa.pdf

[8] Stahl, T., and Völter, M. 2006. *Model-Driven Software Development*, Wiley.

[9] Warmer, J. 2007. *Building a flexible software factory using small DSLs and Small Models*. Presentation at Code Generation 2007, Cambridge, UK.