MetaEdit+: Defining and Using Integrated Domain-Specific Modeling Languages

Juha-Pekka Tolvanen

MetaCase
Ylistonmentie 31
FI-40500 Jyvaskyla, Finland
+358 14 641 000

jpt@metacase.com

Steven Kelly

MetaCase
Ylistonmentie 31
FI-40500 Jyvaskyla, Finland
+358 14 641 000

stevek@metacase.com

Abstract

With MetaEdit+ you can build Domain-Specific Modeling languages and tools — without having to write a single line of code. This demonstration shows how different domain-specific languages (DSLs) can be integrated with high-level metamodels, how languages can be created iteratively while automatically updating existing models, and how multiple modelers can work together seamlessly.

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 Term Design, Economics, Languages

Keywords Domain-specific modeling, domain-specific language, metamodeling, code generation, language workbench

1. Introduction

With MetaEdit+ an experienced developer defines a domain-specific language containing the domain's concepts and rules in a metamodel, and specifies the mapping from that to code in a domain-specific code generator. For the modeling language implementation, MetaEdit+ provides a metamodeling tool suite for defining the language concepts, rules, symbols, checking reports and generators.

Once the metamodel is defined, or even a partial prototype, the rest of the team can start to use it in MetaEdit+. The developers make models with the modeling language and the required code is automatically generated from those models. Based on the metamodel, MetaEdit+ automatically provides full modeling tool functionality such as diagramming editors, browsers, documentation generators, and multi-platform support [1]. MetaEdit+ supports simultaneous multi-user editing via novel locking algorithms [2], as we will demonstrate.

We will show how MetaEdit+ overcomes the problems of other language workbenches by being fast (both for defining and using languages), scalable, and hiding tool implementation details. MetaEdit+ is made with Smalltalk. The demo shows advanced features of creating domain-specific modeling languages, namely language integration, language and model evolution, generator debugging, and different ways to integrate models and code.

OOPSLA 2009, October 25-29, 2009, Orlando, FL, USA.
ACM 978-1-60558-768-4/09/10.

2. Multiple integrated modeling languages

One language is not usually enough: in practice, we need different views or even different languages that can be integrated. MetaEdit+ allows integrated languages to be defined by sharing the same metamodel elements or via explicit integration links. Figure 1 illustrates the latter: a metamodel defines references between models using decomposition or explosion links. These typically support the semantics of top-down abstraction where a design element is linked with another graph that provides a more detailed description of the respective element. Of the two options, decomposition provides stricter semantics by allowing only one subgraph link for each design object. The link also remains the same even if the object is reused somewhere else. Explosion links, on the other hand, are more flexible: several of them can be attached to one element and their scope is limited to one graph only. A given element can thus have a different explosion when it is reused in a new graph, making explosions more like hyperlinks than strict aggregation.

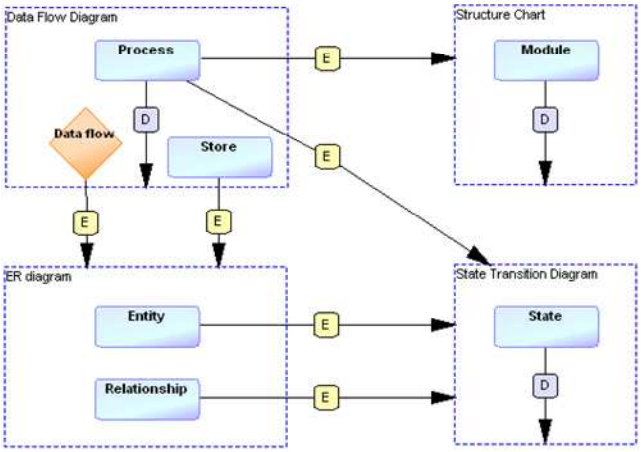

Figure 1. Multiple integrated modeling languages

3. Language and model evolution

When a domain-specific modeling language enters real-life production use, it is inevitable that it will be maintained and modified through its life-cycle — a cycle that may well last for over a decade [3]. The repository of MetaEdit+ offers a natural update policy for sharing the DSM solution: the new language version can be automatically updated for all users and their models. Alternatively, the DSM definition can be shared as an individual file (or files) that modelers can import into their current modeling tool.

It is vital to ensure that information in instance models created with the older version of the metamodel is not lost when the new version is deployed. During the development of MetaEdit+, a lot of effort has been invested in ensuring the seamless updates of metamodels and models. In many cases a conservative approach for modifying the existing metamodels and design data has been adopted. For example, if a concept is removed from the language, the creation of new instances of the type will not be possible, but existing instances of this concept are not removed from the models. Since the generators will still produce working code from these old instances, there is no need to destroy them. Instead, the metamodeler can choose to make them more visibly obsolete, e.g., by changing their symbol to include a red exclamation mark. Checking reports can also be made to list all such obsolete instances, allowing the modeler to make the appropriate update manually. Where the update can be specified, it can also be automated by writing a model transformation. These can be specified in the metamodeler's language of choice operating via the API, or as an XML transformation for the model files.

4. Testing and debugging of generators

In MetaEdit+, generators are defined in the Generator Editor using MERL, a textual DSL designed for turning models into text. Alternative approaches include accessing model data via the SOAP/Webservices API or generating to intermediate files that are then processed by an external program. While DSM best practice is to keep generators simple, on industrial scales generators can still become complex. MetaEdit+ provides a Generator Debugger (Figure 2) to help in tracing and debugging of generators.

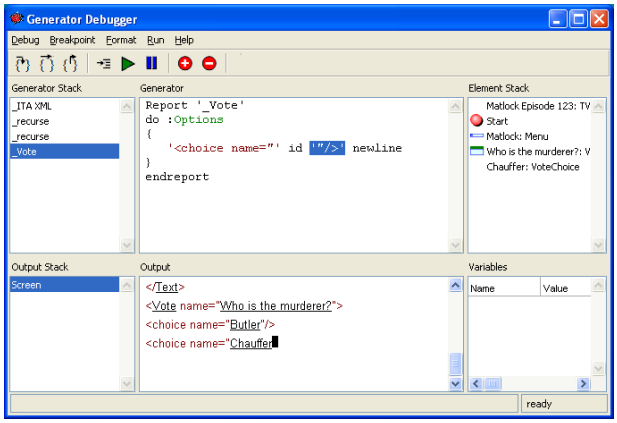

Figure 2. Generator Debugger

The Generator Debugger provides the usual operations for controlling the execution of the generator: setting and removing breakpoints, stepping into/over commands and restarting the execution. The debugger shows several views on the execution status and results. The top half of the window shows the execution status: the call tree of generators, the currently executed generator script, and the stack of model elements that have been accessed to reach the current element. The bottom half of the window shows the rssults: the currently generated target files and output streams, the current output, and the values of the variables used during the generation. The Generator Debugger also offers access to model elements during execution and creates live hyperlinks from the output text back to the model elements.

5. Integrating models and code

There are multiple ways to integrate models and code. We would like to demonstrate one particular feature that is especially good when testing the generator or inspecting the generated code. MetaEdit+ provides a feature called "Live code": a developer may click part of the generated code to jump to the corresponding element in the model. All the generated code provides this capability, so for example in model checking reports and documentation reports it is possible to trace back from the output text to models in MetaEdit+.

6. Summary

Tool support for creating and using modeling languages and code generators is crucial for automating software development. For practical industrial use, the tools must be intuitive and scalable. MetaEdit+ aims to provide this by providing integrated tools for language designers and hiding the tool implementation details. This approach allows the development of domain-specific languages and code generators in a few man-days, not months. Figure 3 illustrates industry experiences in selected domains.

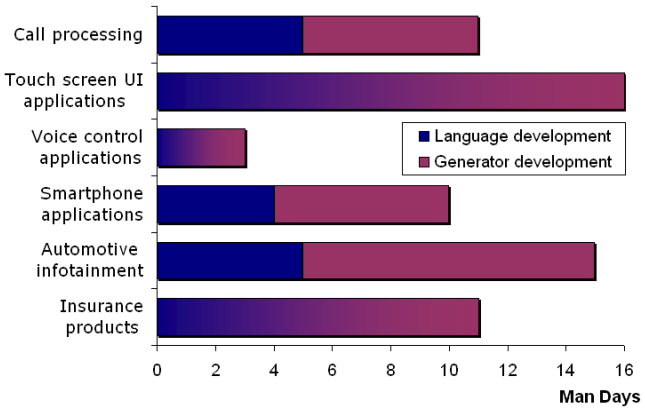

Figure 3. Time to implement modeling tools and generators

The demonstration shows tool features that go beyond editor creation. We demonstrate how multiple languages can be integrated so the changes in models based on one language are visible in other models, offering distinct yet integrated views on the system specified. We also show how the tool supports language evolution, automatically updating existing models to the new version of the language. We show how MetaEdit+ offers debugging tools for both modeling and generators: generated code is linked back to models, so debugging can take place directly in the models. The features are all implemented and available in the free MetaEdit+ evaluation version available from www.metacase.com/download.

References

[1] MetaCase, MetaEdit+ Workbench 4.5 SR1 User's Guide, http://www.metacase.com/support/45/manuals/, 2009

[2] Kelly, S., CASE Tool Support for Co-operative Work in Information System Design, IFIP TC8/WG8.1 Working Conference on Information Systems in the WWW Environment, Beijing, Chapman & Hall, 1998

[3] IEEE Software Special Issue on Domain-Specific Modeling, July/August, 2009