Advanced Tooling for Domain-Specific Modeling: MetaEdit+

Juha-Pekka Tolvanen, Risto Pohjonen, Steven Kelly

MetaCase, Ylistönmäentie 31, FIN-40500 Jyväskylä, Finland
jpt@metacase.com, stevek@metacase.com, rise@metacase.com

Abstract. This paper describes functionalities that go beyond creating the first editor for your modeling language. These are the features that we have found relevant in industrial projects over the past 10 years – all implemented and tested in practice in the MetaEdit+ tool. We demonstrate these advanced tool functionalities from three views, namely from language development view, generator development view, and finally from the DSM use view.

1 Introduction

Tool support for creating and using modeling languages and code generators is crucial for automating software development. Too often the tooling for DSM is focused on creating editors only – often even one editor for one modeling language and for one modeler. This base functionality is available almost in all those over 40 metamodel-based tools developed since the first tool early late 1970's [1]. The further progress in researching and providing tooling support for DSM we need to move beyond these basic functionalities towards those needed in industrial requirements. For example, in industry use it is often more relevant to support the process of DSM creation and evolution than getting the first editor running. In this paper we describe the tooling support we have found useful for defining and using DSM solutions that move beyond initial editor creation. We primarily focus on tooling for defining DSM solutions but also address DSM use.

On the DSM definition side we inspect language definition support, such as:

- Graphical and form-based metamodeling: no programming needed
- WYSIWYG symbol editor with conditional and generated elements
- Integrated, incremental metamodeling and modeling: models update automatically yet non-destructively when the metamodel changes
- Multiple integrated modeling languages
- Multiple simultaneous language developers (metamodelers)

For the generator builder we discuss features, such as:

- Generator editor and debugger integrated with metamodel and models
- Straight model-to-code transformations: no need for intermediate formats
- Generator can map freely between multiple models and multiple files

Finally, with respect to DSM usage we discuss some relevant features, like:

- Multiple simultaneous modelers, free cross-model reuse
- Runs on all major platforms, integrate with any IDE
- "Live code": click generated code to see the original model element

2 Background: MetaEdit+

MetaEdit+ is an integrated, repository-based tool set for creating and using modeling languages and code generators. It was originally developed as a metaCASE tool prototype in the Syti and MetaPHOR research projects at the University of Jyväskylä between 1988 and 1995 [2,3]. The commercial version of the tool has been available from MetaCase since 1993, the latest version at the time of writing being 4.5 from November 2006. MetaEdit+ is currently available for Windows, Mac OS X and Linux operating systems.

MetaEdit+ provides the tool support for different modeling languages by configuring the generic tool set with metamodels. For defining metamodels, MetaEdit+ employs the GOPPRR metamodeling language (Graph-Object-Property-Port-Role-Relationship, [3]). Several modeling languages can be used simultaneously and there can be links and references between different languages. All design data (i.e. metamodels and design model instances made according to them) is stored into an object-oriented repository system which supports complex references between design elements, e.g., inheritance and reuse by reference. The repository also enables multiple users to access and share the design data concurrently. The tool architecture of MetaEdit+ is illustrated in Figure 1.

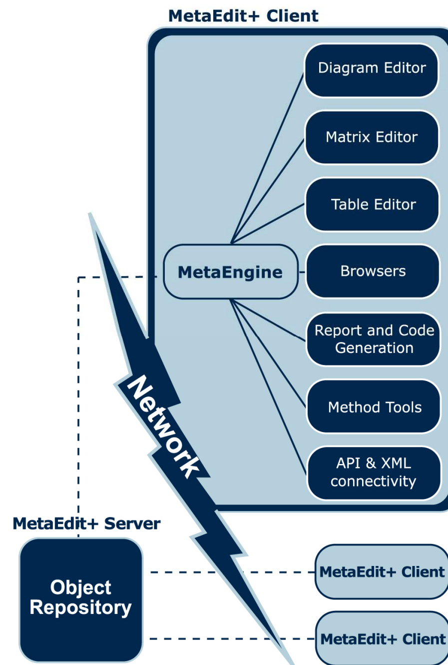

Fig. 1. The tool architecture of MetaEdit+

Models can be accessed and modified by using for four alternative representation styles, namely graphical diagrams, matrixes, tables and tree views. For each format there are different editors and a set of browsers.

3 Tooling features for creating DSM solutions

In MetaEdit+, one or more experts define domain-specific languages as a metamodel containing the domain concepts, rules and notation, and specify the mapping from models to code by defining a domain-specific code generator. This language definition is stored in the MetaEdit+ repository from which it is shared to modelers and can be retrieved for later modification.

3.1 Graphical and form-based metamodeling: no programming needed

In MetaEdit+ languages can be defined in either a graphical or form-based manner using the GOPPRR metamodeling language. A graphical metamodel, as illustrated in Figure 2, is described as a diagram (or a set of diagrams) with the modeling tools of MetaEdit+ using the visual GOPPRR notation [4].

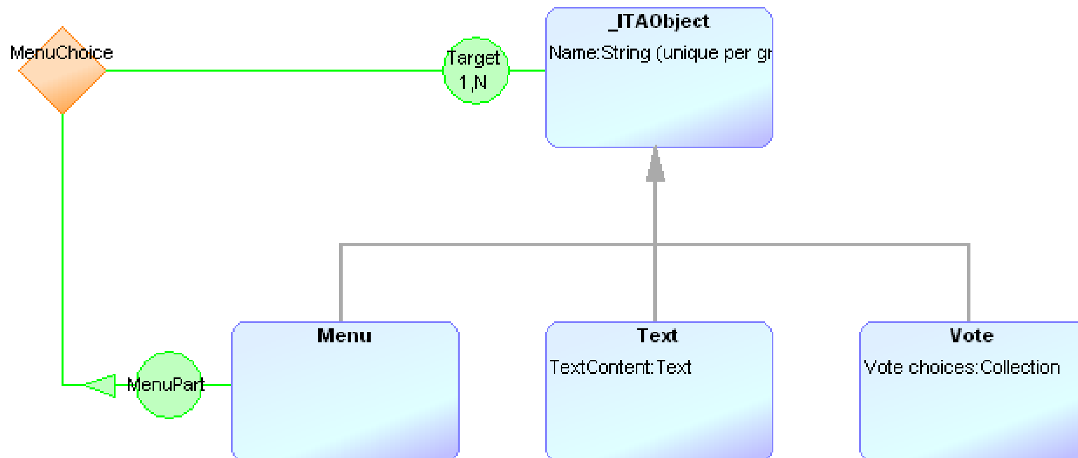

Fig. 2. A graphical metamodel

The metamodel in Figure 2 shows the elementary objects of a modeling language and their combination together with relationships and roles to define how the objects connect to each other. To deploy this graphical metamodel to MetaEdit+, the metamodeler presses the "Build" button on the toolbar. Behind the scenes, this generates an XML document which in turn will be parsed by MetaEdit+ and turned into a modeling language definition in the repository. Once this is done, one can continue finalizing the modeling language with symbol and code generator definitions.

As an alternative to drawing graphical metamodels like Figure 2, MetaEdit+ provides a set of form-based metamodeling tools. These tools are especially useful when modifying a metamodel already in use, or when wanting to inspect the instances of a given metamodel element. A metamodel created graphically can also later be edited with the form-based tools. There is a form-based tool for defining each GOPPRR metatype. An example of an Object Tool with a definition for the Menu object type is shown in Figure 3 (the property ‘Name’ appears red to denote that it has actually been inherited from the _ITAObject ancestor).

Rules and constraints that guide the use of the modeling language are also an important part of the metamodel. In GOPPRR the most elementary set of rules are the bindings that describe how objects, ports, roles and relationships can be combined together to define connection types between objects types. MetaEdit+ also provides means to set constraints on design elements' occurrence, connectivity and uniqueness. Figure 4 shows the Graph Constraints Tool with the definition of a connectivity constraint that limits the number of incoming roles for Menu objects. In addition, other rules are provided like setting default values or defining a regular expression to validate the input values.

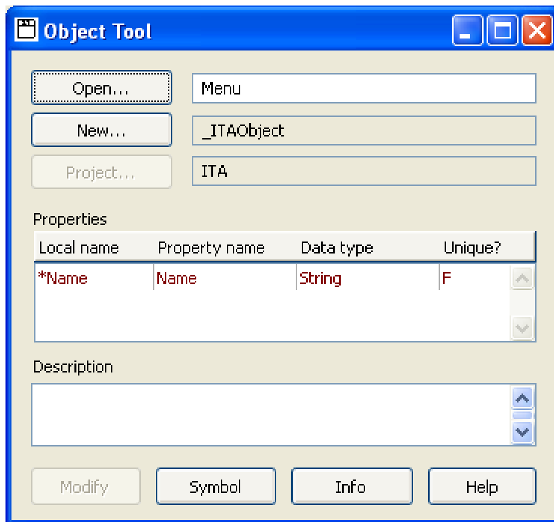

Fig. 3. Object Tool

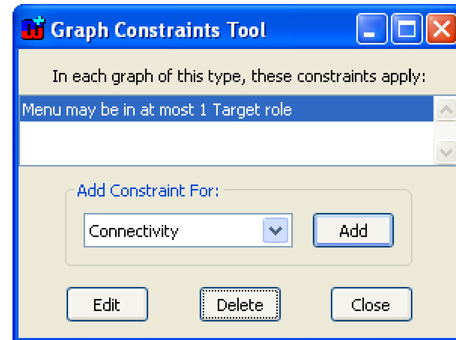

Fig. 4. Graph Constraints Tool

Although the rules are expressed in plain English, they are actually formed by choosing one or more types and integer values in one of a number of rule forms or templates. The rules set by bindings and constraints are enforced automatically and in real-time during modeling. This means that it is not possible to even temporarily create a piece of a model that is not in accordance with the metamodel. However, sometimes such real-time enforcement is not wanted. In such a situation it is better to use generators to create checking reports and execute them when a model validation is needed. Another option is to add optional warning icons to symbols to show when a rule is broken, providing automatic yet unobtrusive visual feedback.

3.2 Symbols with conditional and generated elements

In MetaEdit+, each object, property, port, relationship and role can have a symbol that is used as a graphical representation for that respective concept. Symbols are edited as vector graphics using the Symbol Editor (Figure 5). It is also possible to export and import symbol definitions in SVG format and bitmaps.

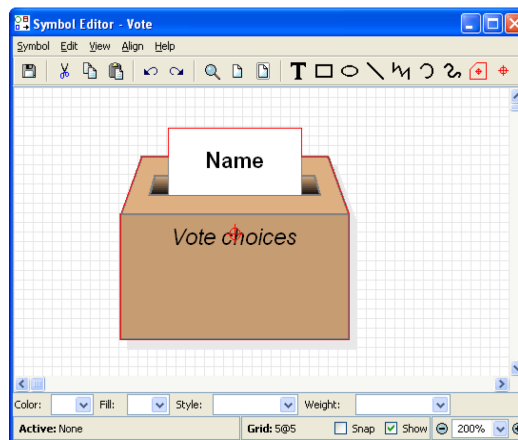

Fig. 5. Symbol Editor

Dynamic behavior of symbol elements is an important aspect to consider when defining symbols. For example, it is quite a typical requirement that certain parts of the symbol may be visible only if certain conditions are met. MetaEdit+ provides extensive support for such conditionality. Each

symbol element can carry a display condition which can be based either on a fixed property value or generation output. Similarly, the content of a text field can be either a fixed text, property value or generator output.

3.3 Sharing the DSM solution and metamodel evolution

If a domain-specific modeling language is considered worthy enough for real-life production use, it is inevitable that it will be maintained and modified through its life-cycle — a cycle that may well reach over a decade. 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

In practice, it is very important to ensure that instance models created with the older version of the metamodel are 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 metatype is removed from the language, no existing instances of this type are removed from the models, but the creation of new instances of the type will not be possible. 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 on the API, or as an XML transformation for the model files.

3.4 Multiple integrated modeling languages

One language is not usually enough: in practice, we need different views or even different languages that can be yet integrated. MetaEdit+ allows defining integrated languages by sharing the same metamodel elements or via explicit integration links. On the latter, references between models can be defined as 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.

3.5 Multiple simultaneous language developers (metamodelers)

Large domains need usually expertise from several persons and many of them can participate also in defining the metamodel. MetaEdit+ supports multiple simultaneous language definers. For this purpose metamodel security can be set to different categories: language definition allowed when models are created, only one person can change the metamodel at a time, or multiple persons can change metamodels. In addition, for each user account metamodeling rights can be set separately.

3.6 Generator editor and debugger integrated with metamodel and models

In MetaEdit+, generators are defined in the Generator Editor using the MERL scripting language. Alternative approaches include accessing model data via SOAP/Webservices API or having intermediate files. These latter two, however, are not integrated with metamodels and therefore their definition and testing takes time and is an unnecessary complex task.

The Generator Editor of MetaEdit+ is the tool for creating generator definitions. It provides many features familiar from full-fledged IDEs like generator definition management tools, a debugger bridge via user-definable breakpoints, syntax checking and text formatting. As for a MetaEdit+ specific feature, the editor also provides shortcuts for the current language concepts – this enables the user to easily refer to and insert the types of the modeling language in the generator script during editing. These elements from the metamodel are shown in different color in the generator than fixed text elements.

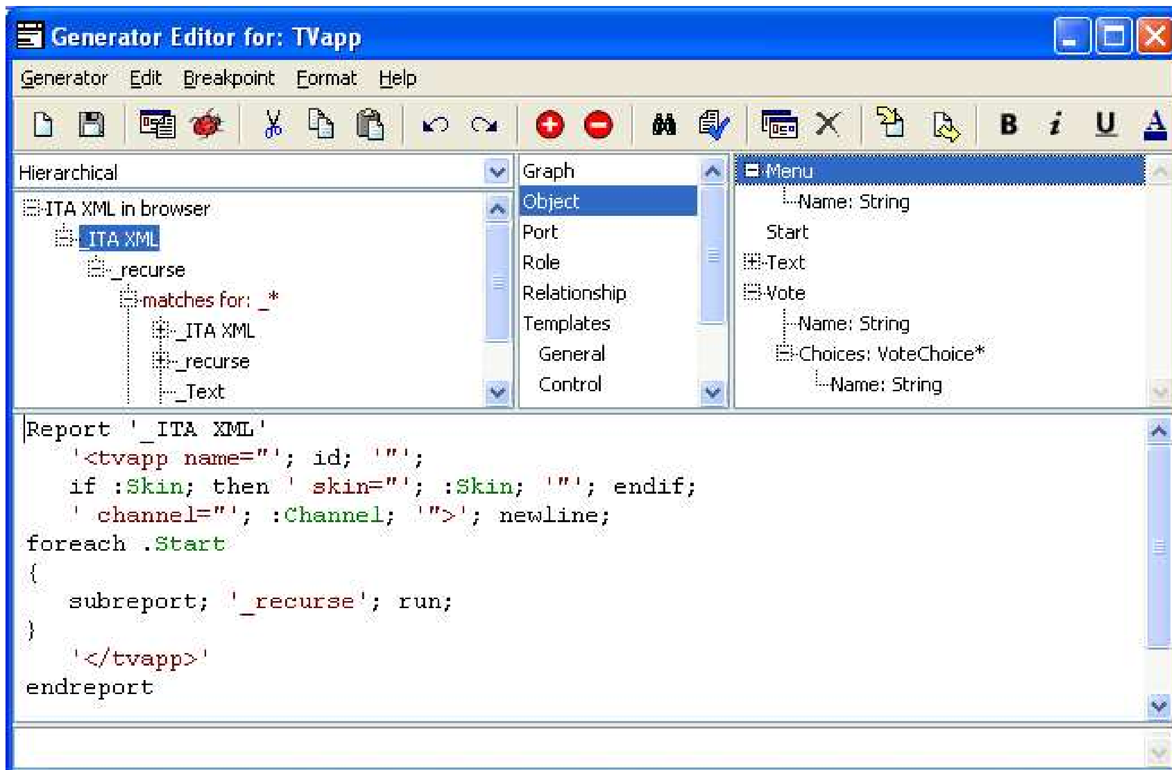

Fig. 5. Generator Editor

Generator definitions are always associated with a certain graph type and are stored with it. It is also possible to associate a generator with the abstract Graph type on the highest level of the inheritance hierarchy. The generator definitions can be inherited: all generators associated with a graph type are automatically available for its descendant types (and in true object-oriented fashion, the descendant graphs can override the generator definitions).

For generator definition, MetaEdit+ uses scripting language called MERL. It is specifically tailored for creating code generation definitions, i.e., domain-specific language for defining generators. It provides powerful means for navigating though the model structures (multiple models and different modeling languages) accessing the design data according to the metamodel. MERL allows translating design data entered in models to the formats required by the generation target language, like remove spaces or use capital letters. MERL can access multiple models and generate multiple files, set protected regions into generated files and can access external files and tools during generation.

3.7 Testing and debugging generators

Real-life generators can still be complex in nature. MetaEdit+ provides a Generator Debugger (Figure 6) for helping in tracing and debugging of generator scripts.

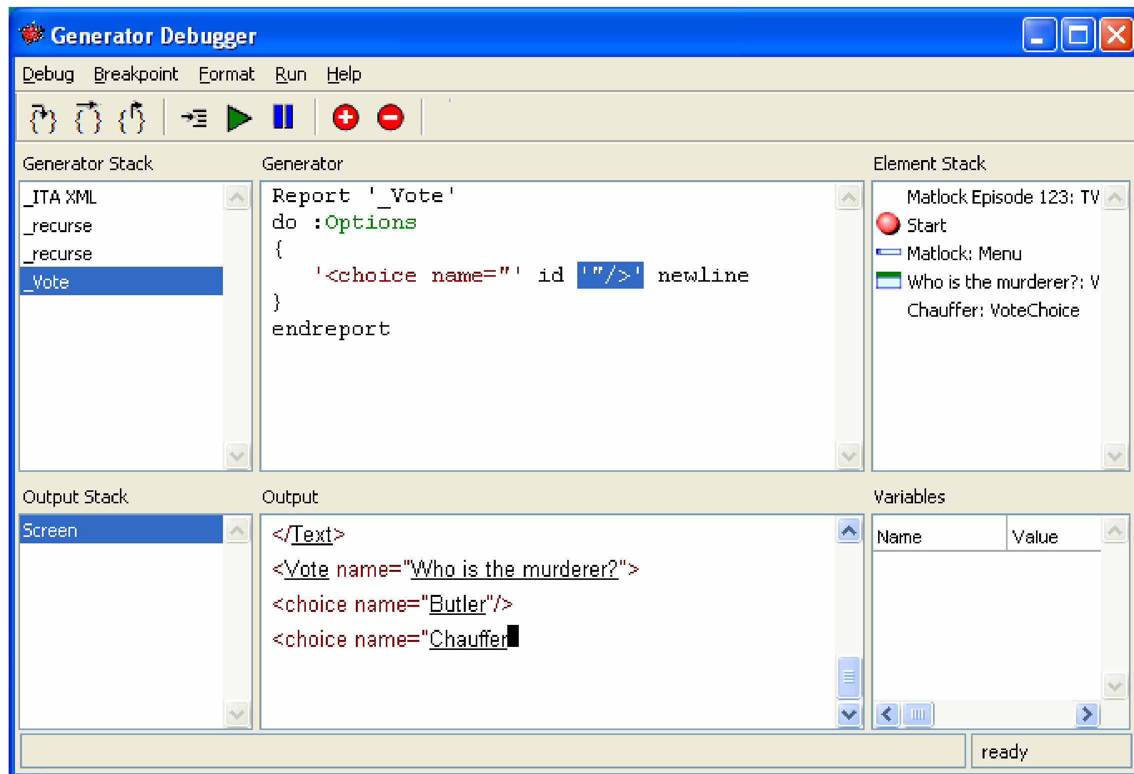

Fig. 6. Generator Debugger

The Generator Debugger provides the usual operations for controlling the execution of the generator script like setting and removing breakpoints, stepping into/over commands and restarting the execution. The debugger shows several views on the execution status and results. Execution status shows the generators run, currently executed generator script and the model elements accessed. Bottom part of the Generator Debugger shows the currently generated targets files and screens, current output, and the value of the variables used during the generation. Generator Debugger also offers access to model elements during execution and gives links from produced output back to the respective model element.

4 Tooling features for DSM use

4.1 Multi-user environment

As mentioned earlier, the object repository employed by MetaEdit+ enables multiple users to access the design data concurrently. This opens up an avenue for sophisticated ways for the division of labor. For example, one modeler may change a model element in one design and the change is automatically reflected to those modelers that have the same model element. Also, refactoring of models became easier since the model data can be changed only in one place.

The multi-user features also bring up the question about concurrency control. In MetaEdit+ concurrency is managed with the transaction and locking system provided by the object repository. Each user see the changes others have committed to the repository and when the user commits his or her changes, they become available for others. The locking mechanism prevents

editing conflicts between users: when one user is editing a model, others may view but not change it. MetaEdit+ provides several levels of granularity for the locking, enabling the users not only to get the relevant locks but also to avoid locking design data unnecessarily.

4.2 Platform dependency

MetaEdit+ is made to work on all major platforms and integrate with other tools in the chain. Alternative tool integration approaches include:

- Programmatic access to model data and MetaEdit+ functions via API
- Model importing and exporting as XML
- Command line parameters for automating MetaEdit+ operations
- Executing external commands via generators

4.3 Integrating models and code

There are multiple ways to integrating code and models. 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": developer may click generated code to see the original model element. All the generated code provides this capability so for example in model checking reports and documentation reports it is possible to trace back from produced text back to models in MetaEdit+.

5 Conclusions

This paper described some of the features for DSM tooling that go beyond editor creation. The features all are implemented and available in MetaEdit+ tool. Our objective is to demonstrate them during the 7th OOPSLA workshop on Domain-Specific Modeling. We believe this will raise interesting discussion for further improving tooling support for DSM.

References

1. Teichroew, D., Macasovic, P., Hershey, E., Yamamoto, Y., Application of the entity-relationship approach to information processing systems modeling. In Entity-Relationship Approach to Systems Analysis and Design, P. P. Chen (Ed.), Amsterdam: North-Holland, 1980.
2. Smolander, K., Lyytinen, K., Tahvanainen, V.-P., and Marttiin, P., "MetaEdit: A flexible graphical environment for methodology modelling", Proceedings of CAiSE'91, 3rd Intl. Conference on Advanced Information Systems Engineering, Springer Verlag, pp. 168-193, 1991.
3. Kelly, S., Lyytinen, K., and Rossi, M., "MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE Environment”, Proceedings of CAiSE'96, 8th Intl. Conference on Advanced Information Systems Engineering, Lecture Notes in Computer Science 1080, Springer-Verlag, pp. 1-21, 1996.
4. MetaEdit+ 4.5 User’s Manuals, MetaCase, November 2006 (http://www.metacase.com/support/45/manuals/index.html)