Interactive Television Applications using MetaEdit+

Risto Pohjonen
MetaCase
Ylistönmäentie 31
FI-40500, Jyväskylä, Finland
rise@metacase.com

Steven Kelly
MetaCase
Ylistönmäentie 31
FI-40500, Jyväskylä, Finland
stevek@metacase.com

ABSTRACT
This paper discusses the use of MetaEdit+ as a tool for creating and using domain specific modeling languages and code generators. Creating a modeling language with graphical and form-based metamodeling, the usage of MetaEdit+'s modeling tools and the development of a code generation are demonstrated using an Interactive Television Application example. Further tooling issues, such as integration and language evolution, are also addressed at the end of the paper.

General Terms
Design, Languages

Keywords
Domain-specific modeling, tool support, MetaEdit+

1. INTRODUCTION
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 [1,2]. 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 these metamodels MetaEdit+ employs the GOPPRR metamodeling language [2]. 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.

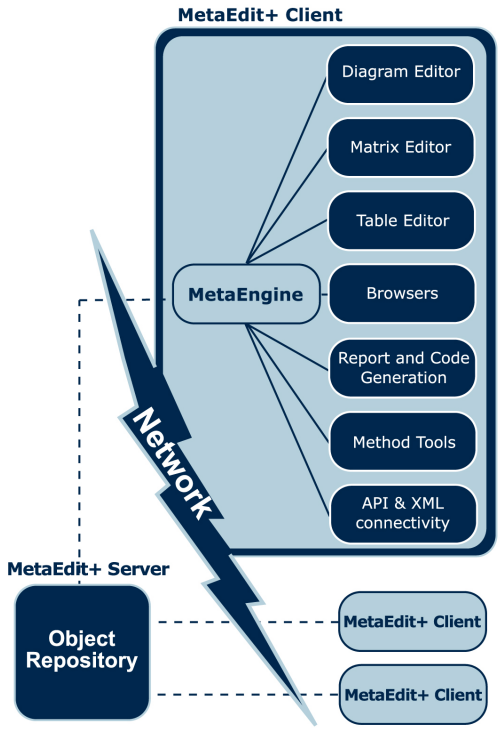

Figure 1. The tool architecture of MetaEdit+

MetaEdit+ comes in two versions. MetaEdit+ Workbench (MWB) integrates the language and generator development tools and ordinary modeling tools, whereas the plain client version of MetaEdit+ includes only the ordinary modeling tools.

2. LANGUAGE CREATION
This section describes how a domain-specific modeling language and its tool support can be developed with MetaEdit+. The development process, which includes the creation of the metamodel and the definition of the language constraints and rules as well as the visual notation, is explained using the Interactive Television Application (ITA) domain as an example [3].

2.1 Metamodel
As mentioned earlier, MetaEdit+ uses metamodels to configure its tool behavior for any specific modeling language. Metamodels can be defined in either a graphical or form-based manner using

the GOPPRR metamodeling language. The name GOPPRR is an acronym made out of the names of the metatypes recognized by the language: **G**raph, **O**bject, **P**roperty, **P**ort, **R**elationship and **R**ole. A Graph is stereotypically a diagram, consisting of Objects connected by Relationships. To be more precise, the center point or hub of a set of connecting lines is a Relationship (e.g. the meeting point of the gray lines in Figure 2), and each line from there to an Object is a Role (e.g. there are three subclass Roles attached to that relationship, and one superclass Role with an arrowhead). An Object may also have specific Ports on its rim to which Roles may connect; different Ports would have different semantics. All of the above may have Properties specifying their names and other information; a Property value can also be any of the above, allowing complex structures of arbitrary depth.

2.1.1 Graphical metamodeling

A graphical metamodel is described as a diagram (or a set of diagrams) with the modeling tools of MetaEdit+ using the visual GOPPRR notation [4]. The example of the graphical metamodel for the ITA modeling language is presented in Figure 2.

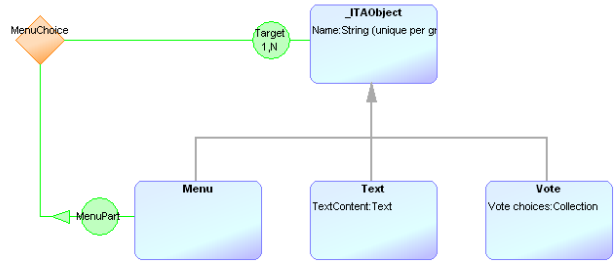

Figure 2. A graphical metamodel for ITA example

The metamodel in Figure 2 shows the elementary objects of the ITA modeling language and their combination together with relationships and roles to define how the objects connect to each other. The abstract object _ITAObject serves two purposes here. First, it provides the common property of Name for all its descendants. Second, it simplifies the definition of the MenuChoice binding. We want to semantically define that there can be a link from one Menu to another Menu or Text or Vote. We could of course always define three explicit bindings for each scenario (Menu – Menu, Menu – Text and Menu – Vote), but with _ITAObject, we can now define the binding in more general manner by saying "a menu can connect to any descendant of _ITAObject".

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.

2.1.2 Form-based metamodeling

As an alternative to drawing graphical metamodels like Figure 2, MetaEdit+ provides a set of form-based metamodeling tools. Using these tools may be less intuitive for beginners, but they provide more precision, some extra options and in our experience better scalability than graphical metamodels. 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).

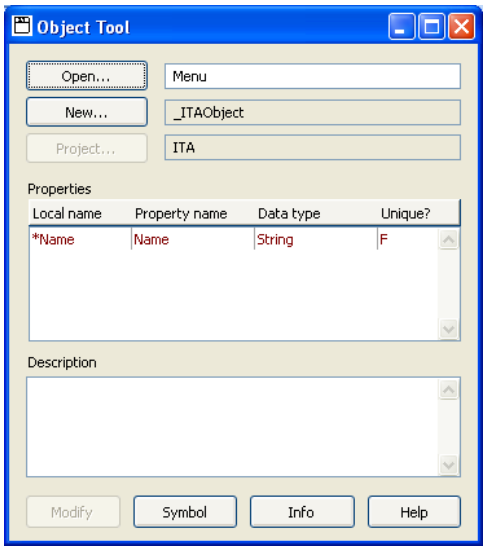

Figure 3. Object Tool

Each object, port, role and relationship type can be defined in this way. The properties attached to these types are defined with the Property Tool (examples shown in Figure 4). All property types must have a name and a data type defined. Depending on the data type, there are also additional definitions. The Property Tool on the left side of Figure 4 defines a string property 'Name' which the modeler will enter with a normal input field widget. Other widgets could be used, e.g. a pull-down list, combo box or radio button set. It is also possible to set a default value and a regular expression to validate the input value. The Property Tool on the right side, on the other hand, defines a property type for vote choices. In this case, the data type is a collection of string elements. There is no need to set widget and default value, but it is still possible to define the validation regular expression.

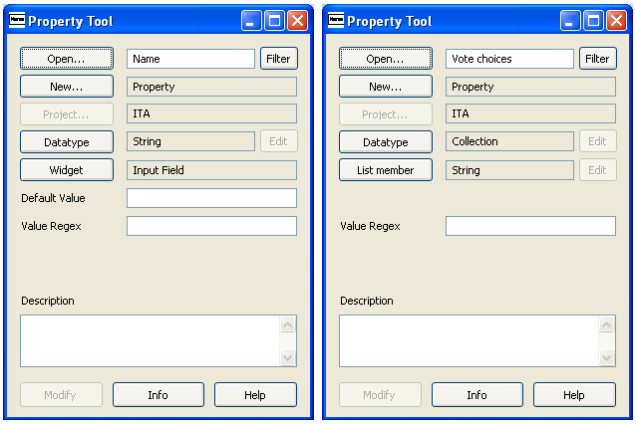

Figure 4. Property Tools

Once the individual types and their properties have been defined, the final language definition can be put together with the Graph Tool (Figure 5).

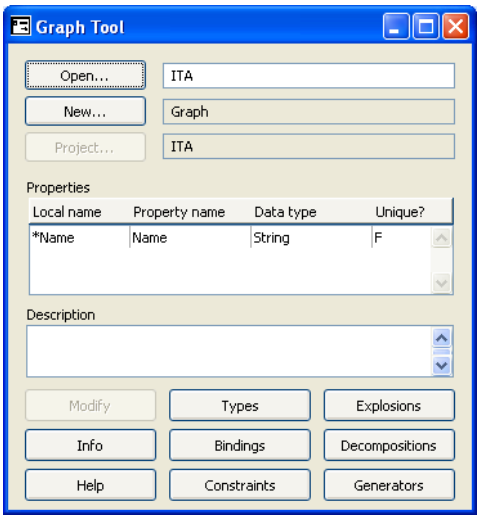

Figure 5. Graph Tool

As the top-level modeling language editor, the Graph Tool combines the individual types into a meaningful whole. The first thing to do is to pick the types for the language and describe how they participate in bindings: how each relationship type may connect certain types of objects in certain types of roles. An example of the binding definition for the MenuChoice relationship type is shown in the Graph Bindings Tool in Figure 6.

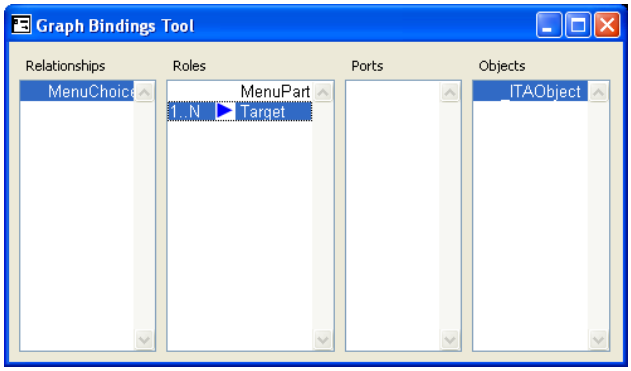

Figure 6. Binding definition

In addition to bindings, it is also possible to set constraints for the language elements (see Section 2.2) or define references between models and model elements.

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 7 shows the Graph Constraints Tool with the definition of a connectivity constraint that limits the number of incoming roles for Menu objects.

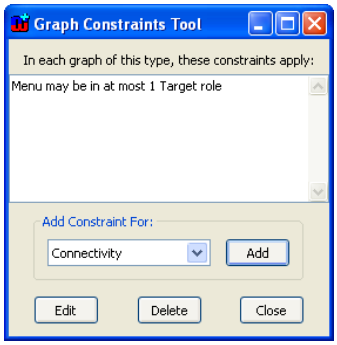

Figure 7. Graph Constraints Tool

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 model that is not in accordance with the metamodel (for example, it is not possible to draw a MenuChoice relationship between two Vote objects). However, sometimes such real-time enforcement is not wanted. In such a situation it is better to use generators to create checking reports for models 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.

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.

The definition of bindings, constraints and sub-graph links within the Graph Tool completes the form-based metamodeling. As with graphical metamodeling, the work continues now with symbol definition and code generator development.

2.1.3 Choosing the right metamodeling approach

According to our experience, the graphical metamodels are best suited to smaller modeling languages, and for less experienced metamodelers. Graphical metamodels also provide a good way to document modeling languages. For production-level modeling languages it is better to use form-based metamodeling. This is mainly because the form-based metamodeling provides better support for layered structures and complex references between design elements often found in such languages.

The form-based metamodeling also has one major advantage over the graphical one: the bridge between the metamodel and model instances is live. This enables the metamodeler to test the language definition instantly in the modeling tools of MetaEdit+ while creating the metamodel with the metamodeling tools.

2.2 Symbols

No metamodel is complete unless it defines the visualization for the incorporated language concepts. In MetaEdit+, each object, 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 8). It is also possible to export and import symbol definitions in SVG format.

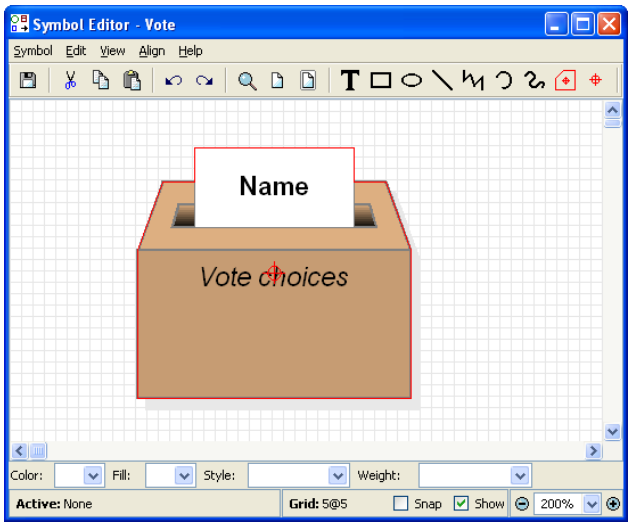

Figure 8. Symbol Editor

MetaEdit+ also enables the use of bitmaps as symbol elements. However, according to our experience, even if the bitmaps can be used nicely to spice up the symbols, exclusive use of bitmaps should be avoided. The major argument against bitmaps is their poor scalability – it is very likely that the symbols will be scaled to different proportions and aspect ratios in the instance models.

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. An example of the last case is visible in the ITA language: the menu options shown within the Menu symbol are actually generated by looking at how the menu is connected to sub-elements.

With our symbols defined, it is now time to see how the newly created language can support modeling in MetaEdit+

3. MODELING

Modeling in MetaEdit+ is carried out in the Diagram, Matrix and Table Editors. Each of these three editors provides a different representational view of the underlying conceptual data. In addition to these editing tools, there are also support tools for browsing and finding design elements.

3.1 Diagram Editor

The Diagram Editor is the main vehicle for modeling in MetaEdit+. This is simply because as a provider of the graphical representation of design data, it is the natural choice for working with visual modeling languages. As shown in Figure 9, the Diagram Editor offers a traditional graphical editing tool. The design elements, represented by their symbols, can be freely placed on the drawing area and they can be controlled with typical graphical editing operations (e.g. moving, scaling, connecting, etc.). The properties within the elements can be accessed either directly through dialogs or from the sidebar on the left.

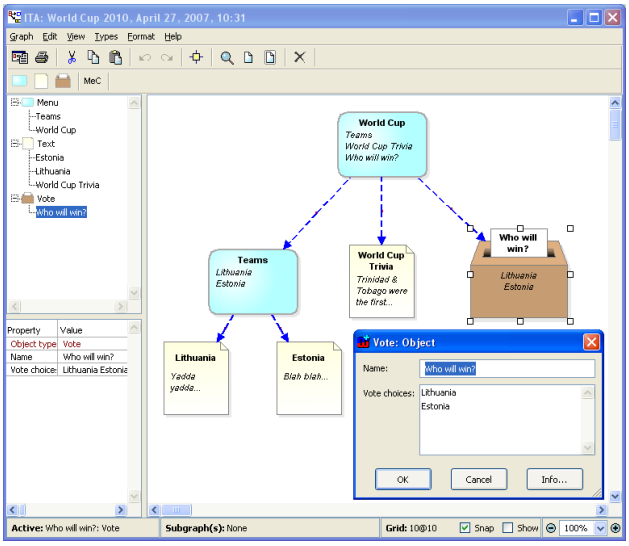

Figure 9. Diagram Editor

The Diagram Editor also provides the means for filtering the view as well as the possibility to export the diagram to the clipboard or to various graphical file formats. Code generators can be also executed for the currently opened model from the Diagram Editor (and from other editors as well).

3.2 Matrix and Table Editors

In addition to the graphical view shown by the Diagram Editor, MetaEdit+ also provides other options for viewing data. The Matrix Editor (shown in Figure 10) offers a matrix-based view with the emphasis on the relationships between objects. The Matrix Editor has several filtering options for customizing the view and it also provides tools for matrix analysis operations like sorting and diagonalizing. The Matrix Editor is a useful tool for accessing and editing models with large numbers of relationships.

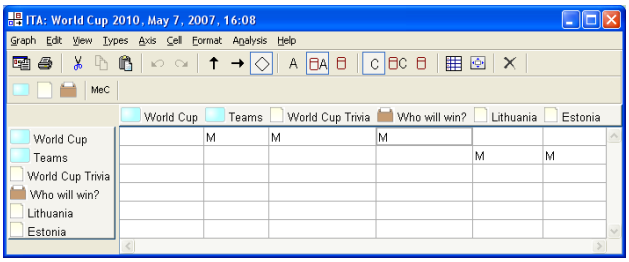

Figure 10. Matrix Editor

The Table Editor (Figure 11) puts the emphasis on objects and their properties. Its main advantage over the Diagram or Matrix Editors is in its ability to provide a distilled view of a set of design elements with all their properties. The rows represent objects and the columns properties, and thus all objects in a table must be of

the same type. As relationships are not visible, this also makes the Table Editor a possible solution to Use Case 3: only the Text objects can be edited from here. With the multi-user locking of MetaEdit+, the producer could still edit the structure of the application and its other elements in a Diagram Editor, while the journalist could edit the contents of the texts in a Table Editor.

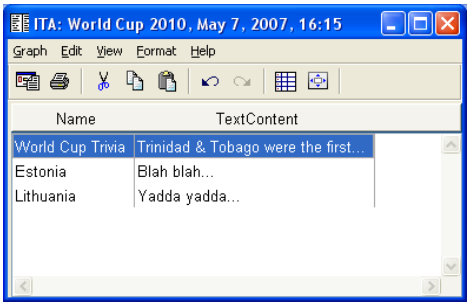

Figure 11. Table Editor

3.3 Browsing and finding design elements

Real-life modeling work also requires the abilities to browse and search the design elements. Quite often this is related to the reuse of the elements: it is important either to be able to browse what kinds of elements already exist or to be able to find out where a specific element might already be in use.

For browsing the existing elements, MetaEdit+ offers four browsers, each of them providing a different starting point and hierarchy for accessing the elements. For example, a Graph Browser shows how design elements are structured according to the graphs, whereas the Metamodel Browser shows how different types are organized under the respective metamodels. The views provided by the browsers can be customized by user-definable filters. The various browsers can be accessed from the main MetaEdit+ launcher window (Figure 12).

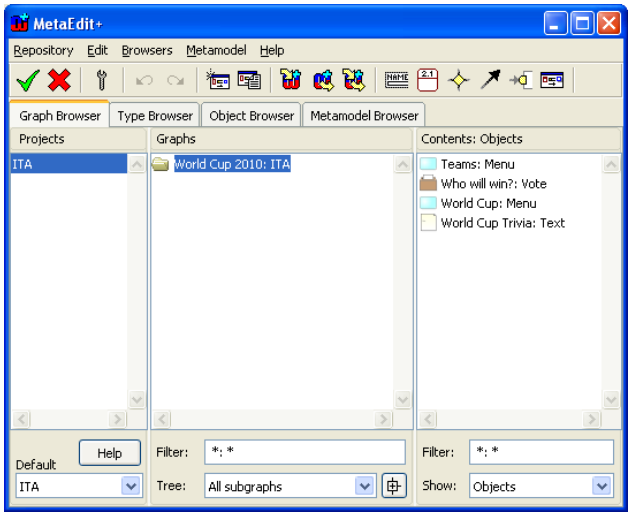

Figure 12. MetaEdit+ main launcher with Graph Browser opened

MetaEdit+ can also show the places where a specific design element has been used. This can be done by picking up a design element and opening the Info Tool for it. Figure 13 shows an Info Tool opened for the 'World Cup' Menu object.

Figure 13. Info Tool

3.4 Multi-user environment

As mentioned earlier, the object repository employed by MetaEdit+ can enable multiple users to access the design data concurrently. This opens up an avenue for sophisticated ways for the division of labor. For example, for Use Case 3 of the ITA example, the multi-user environment provides a viable solution: the journalist reporting the rugby match can change the relevant part of the model while the producer retains control over the whole application.

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 can 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. For example, considering again Use Case 3 mentioned above, the following locking scenario can be used: the producer locks the main application model, but leaves the rugby match info page objects unlocked so that the journalist can access and change them later.

4. CODE GENERATION

The idea of code generation is simple: the generator crawls through the design models, extracts design data from them and outputs it in some predefined format. In this chapter we discuss how the generators work and are defined in MetaEdit+.

4.1 Executing generators

Code and other generators can be launched from the MetaEdit+ main launcher or from the individual editors. The user picks the generator from the list of available generators and this generator will be executed for the currently selected or opened model. The output will be forwarded either to the screen or file. Depending on the generator, it is also possible to carry on additional tasks during or after the actual code generation, like compiling and executing the generated code or starting other external programs. Once the generation is completed, MetaEdit+ will show a report of generation results. An example of code generation output for the model presented in Figure 9 is shown below:

```
<TVApp name="World Cup 2010">
  <Menu name="World Cup">
    <Menu name="Teams">
      <Text name="Estonia">Blah blah...</Text>
      <Text name="Lithuania">Yadda yadda...</Text>
    </Menu>
    <Text name="World Cup Trivia">
      Trinidad &amp;amp; Tobago were the first...
    </Text>
    <Vote name="Who will win?">
      <Choice name="Lithuania"/>
      <Choice name="Estonia"/>
    </Vote>
  </Menu>
</TVApp>
```

4.2 Creating generators

In MetaEdit+ generators are defined in the Generator Editor using the MERL scripting language.

4.2.1 Generator Editor

The Generator Editor (shown in Figure 14) 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 green in the generator; fixed text elements are shown in red.

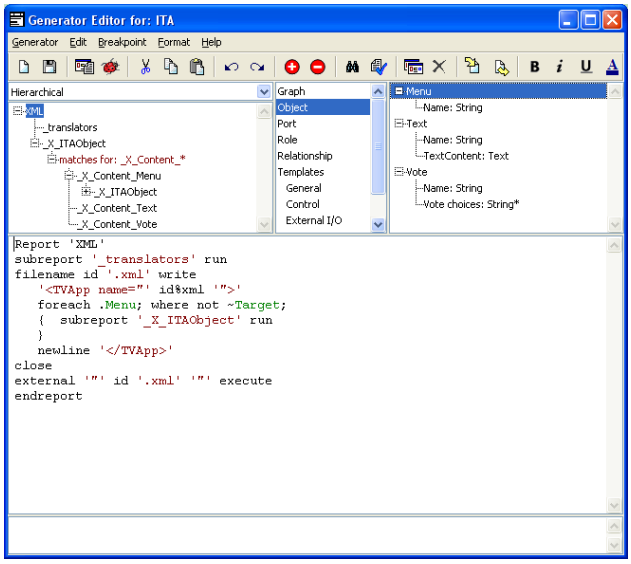

Figure 14. Generator Editor

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).

4.2.2 MERL generator definition language

The MERL scripting language is specifically tailored for creating code generation definitions. It provides easy yet powerful means for navigating though the model structures and accessing the design data according to the metamodel, which is an elementary requirement for efficient domain-specific code generators. For each generation the starting context for the generator is defined by the currently selected or opened model, and the scope broadens during generation by following the relationships and sub-graph links within models.

Figure 15 illustrates the structure of the XML generator for the ITA example. As can be seen, the generator is implemented in modular fashion which makes it easier to maintain and modify the generator during its life-cycle.

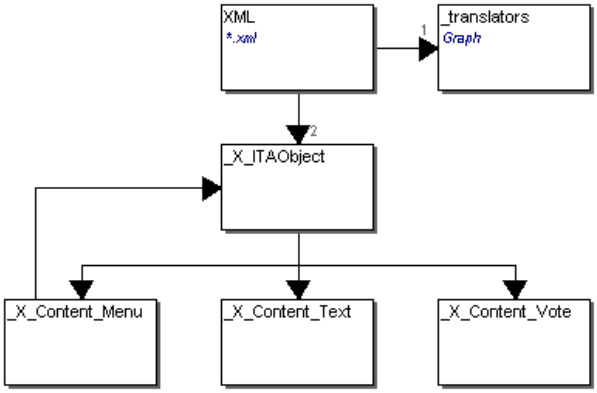

Figure 15. The ITA generator structure

The definition for the top-level generator module ‘XML’ is presented below. For ease of reading, fixed text output directly to the XML file is shown with a gray background.

```
01  Report 'XML'
02  subreport '_translators' run
03  filename id '.xml' write
04      '<TVApp name="' id%xml '">'
05      foreach .Menu; where not ~Target;
06      {       subreport '_X_ITAObject' run
07      }
08      newline '</TVApp>'
09  close
10  external '"' id '.xml' '"' execute
11  endreport
```

Let us look first at the window dressing in the first and last couple of lines. The sub-generator call to ‘_translators’ on line 2 loads a predefined set of translation macros for converting troublesome characters like &, < and > to XML compliant form. This can be seen in use in line 4, where `id%xml` means “output the identifying property of the current element, filtered by the xml translation”. The `external...execute` call in line 10 will open the resulting XML file in the default application, e.g. Internet Explorer.

The main body of this generator takes place in lines 3 to 9. The `filename...write...close` structure directs the output to a file. On line 3, all output between `filename` and `write` goes to build up the filename. The keyword `id` calls for the identifying property of the graph on which the generator is executed. Appending the fixed string '.xml' results in a file with the same name as the model, with extension .xml. All output between `write` and `close`, i.e. lines 4–8, will be directed to that file.

Line 5 says to start from each Menu which does not have an incoming relationship — i.e. the topmost menu — and line 6 runs the sub-generator called _X_ITAObject for this menu. At this point, in the beginning of the '_X_Menu' generator module, it is important to notice that the generation content has changed: we are no longer within the broad context of the graph but within the context of the currently accessed Menu object in that graph.

```
01  Report '_X_ITAObject'
02  newline '<' type ' name="' id%xml '">'
03  to '%indent' translate
04      subreport '_X_Content_' type run
05  endto
06  newline '</' type '>'
07  endreport
```

The _X_ITAObject generator handles all the parts of the generation that are the same for the different kinds of elements. The main similarities are the opening and closing tags in lines 2 and 6. In line 2, the `type` keyword is used to fill in the tag type from the element's type (Menu, Vote or Text). The name attribute is retrieved with `id`, which in this case of course refers to the identifier of the currently accessed object. Note again the use of `%xml` translator. Between lines 3 and 5, all output will be indented by one level by the %indent translation. This simply replaces each newline which is output between `translate` and `endto` with a newline plus a tab.

The actual content of each tag differs depending on the type of the element, so line 4 calls a sub-generator whose name is built dynamically from _X_Content_ plus the name of the type, e.g. _X_Content_Menu.

```
01  Report '_X_Content_Menu'
02  do ~MenuPart~Target.()
03  orderby x num
04  {   subreport '_X_ITAObject' run
05  }
06  endreport
```

From the _X_Content_Menu sub-generator we notice that Menus themselves actually have no content of their own: they simply contain the elements they link to with relationships. These relationships form the basis of recursion of the generation. The `do` loop on line 2 instructs the generator to reach all objects that can be accessed by following a MenuPart role attached to the current Menu, crawling through the relationship to a Target role at the other end, and on into the attached object (the empty parentheses simply mean to allow any kind of object). Line 3 sorts the menu contents based on their horizontal position. Within the `do` loop the generation proceeds in the context of the object reached with this navigation. So, when the sub-generator call on line 4 is made, the _X_ITAObject sub-generator is called in the context of a sub-element of the Menu, e.g. a Text. The '_ITAObject' generator creates the opening tag for the now-current object, e.g. a Text, adds a level of indentation, and calls the appropriate _X_Content_* sub-generator, e.g. _X_Content_Text for the content of the Text object:

```
01  Report '_X_Content_Text'
02  :TextContent%xml
03  endreport
```

As we can see, the content generated for a Text is just a single element: the TextContent property's value, filtered by the xml translation. If the sub-element had been a Vote, the sub-generator would have been _X_Content_Vote:

```
01  Report '_X_Content_Vote'
02  do :Vote choices
03  {   newline '<Choice name="' id%xml '"/>'
04  }
05  endreport
```

The loop between lines 2 and 4 iterates though the `Vote choices` collection and outputs the Choice tags. The `id` keyword this time refers to each individual string within the collection.

For completeness, here are the %xml and %indent definitions from _translators:

```
01  Report '_translators'
02  /* translate to legal XML text or value */
03  to '%xml
04  & $&amp;
05  < $&lt;
06  > $&gt;
07  " $&quot;'
08  endto
09
10  /* indent one level: newline->newline+tab */
11  to '%indent
12  /(\r|\n)/ $$1      '
13  endto
14  endreport
```

Each line between a `to` and `endto` consists of the left and right hand sides of a mapping, separated by a space. The %xml translator in lines 4 to 7 shows how each occurrence of a single character can be translated to a string — strings are prefixed by $. The %indent translator in line 12 shows how a regular expression — surrounded by slashes, / — can be translated to a string, prefixed by $. The string can include the matches to parenthesized

parts of the regular expression, e.g. $1. Other possibilities include ranges, e.g. %upper is simply ‘a-z A-Z’; and defaults, e.g. to turn all non-alphanumerics into underscores, ‘a-zA-Z0-9 *’ keeps alphanumerics unchanged, while ‘* _’ maps any remaining characters to underscore.

4.2.3 Generator Debugger

According to our experience, one should always try to keep the generators as simple as possible. All possible dealing with variation should be excluded from the generator, either by passing it to the modeling language or providing better level of abstraction on the code side with frameworks or components. Despite these good efforts and emphasis on modularization and reuse in generator development, real-life generators can still be complex in nature. Therefore it is vital that good support tools exist for generator development as well. In addition to the previously mentioned Generator Editor, MetaEdit+ also provides a Generator Debugger (Figure 16) for helping in tracing and debugging of generator scripts.

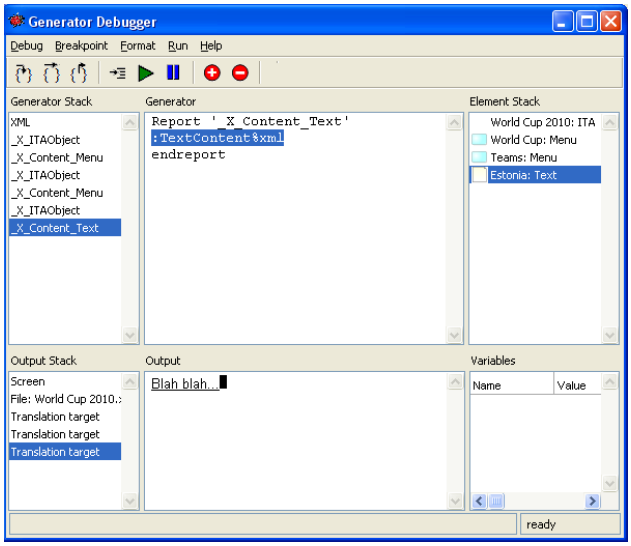

Figure 16. Generator Debugger

The Generator Debugger provides the usual operations for controlling the execution of the generator script like setting and removing breakpoints, stepping commands one by one and restarting the execution. The debugger also provides several views on the execution status and results. At the top left is the generator stack, showing the initial invocation of the XML generator at the top, and the current _X_Content_Text at the bottom. At the top center is the running generator’s content. At the top right is the Element Stack, showing the elements that have been navigated through to reach the current element. Here we started with the initial World Cup 2010 graph and descended through two Menus to the current Text object, Estonia. At the bottom left is the Output Stack, showing the streams that are being output to, along with any stream filtering caused by translations. In this case there are three translations, one for each %indent as we passed through _X_ITAObject. The %xml translation has already been applied, resulting in the text shown in the Output pane. At the bottom right is a list of all variables and their values; this example, like many, does not require variables.

5. ADVANCED TOOL SUPPORT ISSUES

After the completion and deployment of the domain-specific modeling language, there are typically a number of further tool support issues that will arise when the language is taken into production use. Typical such questions, like integration with other tools and language evolution, are discussed in this section.

5.1 Tool integration

In addition to language development, modeling and generator tools MetaEdit+ also provides a set of tools and technologies for integrating the tool with others. The API opens the conceptual and representational design data for external users and provides an interface for accessing, retrieving and creating design elements in MetaEdit+. There is also some additional functionality provided by the API interface, like the possibility to highlight design elements in MetaEdit+ Diagram Editor for tracing and simulation purposes. The other viable option for integration is the XML export and import feature which works for metamodels, models and representations.

5.2 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 the ten years anniversary mark. With metamodel evolution 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.

6. CONCLUSION

In this paper we have demonstrated how the development of a domain-specific modeling language and its tool support (including code generation) can be carried out with MetaEdit+, using the ITA domain as an example. As a conclusion, we would like to go back to the original ITA assignment and re-evaluate the presented use cases from the point of view of the developed language and tool support provided by MetaEdit+.

Case 1: As Figure 9 illustrates, this use case can be supported to the detail with the solution sketched in this paper.

Case 2: The producer could draw the structure of the application, with one Menu per group. The team information could be prepared as Text objects, either in the diagram but not yet

connected by relationships to groups, or then in a separate Table on the same graph. When the assignment to groups is known, the teams for a group could be picked or cut from the Table, then connected by relationships. A relationship from a group Menu to all its team Texts can be drawn in a single action, by selecting the Menu then dragging to select the Texts. Finally, the automatic Layout algorithm could be applied to beautify the resulting structure.

Case 3: This use case was covered in the Table Editor, Section 3.2, Figure 11.

Case 4: This use case is apparently more about tool support for the version control than about the language itself. Depending on the situation, MetaEdit+ can provide several solutions. At the finest level of granularity, individual operations can be reverted using the unlimited Undo buffer. As the MetaEdit+ repository system is transaction based, the next simplest solution would be to abandon the current transaction and roll-back to the state at the end of the previous transaction. Solutions for broader levels of granularity depend on the version control system in use. MetaEdit+ can support check-in/check-out for various systems with the XML export/import or API link.

As for the possible directions for the future development of the ITA language, we would like to emphasize the reuse and linking of the application content. For instance, in the World Cup example application it would make sense to be able to link the vote choice with the text pages that provide the information about each team: the same set of teams will be present in both. This would require a new relationship type to link a Vote to an existing Menu, giving that Vote the same choices as in the menu. The only change necessary to the generators would be to extent _X_Content_Text with navigation through that relationship and iteration over the Menu choices, e.g.

```
05  do >Share.Menu~MenuPart~Target.()
06  {   newline '<Choice name="' id%xml '"/>'
07  }
```

7. REFERENCES

[1] 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.

[2] 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.

[3] www.dsmforum.org/events/MDD-TIF07/InteractiveTVApps.pdf

[4] Graphical Metamodeling Example. MetaCase, November 2006