An Introduction to the Generic Modeling Environment

Zoltán Molnár
Vanderbilt University
Box 351829, Station B
Nashville, TN 37235
+1-615-322-8747
zolmol@isis.vanderbilt.edu

Daniel Balasubramanian
Vanderbilt University
Box 351829, Station B
Nashville, TN 37235
+1-615-343-7472
daniel@isis.vanderbilt.edu

Ákos Lédeczi
Vanderbilt University
Box 351829, Station B
Nashville, TN 37235
+1-615-343-8307
akos@isis.vanderbilt.edu

ABSTRACT

In this paper, we describe the Generic Modeling Environment (GME), a configurable graphical modeling toolsuite that supports the rapid creation of domain specific modeling, model analysis and program synthesis environments. The metamodeling, modeling and code generation features are illustrated through a case-study.

Keywords

GME, Generic Modeling Environment, Domain Specific Modeling Language, DSML, Model Integrated Computing, MIC

1. INTRODUCTION

Model Integrated Computing (MIC) has been developed at Vanderbilt University as a methodology for building embedded software systems utilizing domain-specific modeling environments [2]. MIC is a way to develop systems while addressing problems of system integration and evolution. MIC is used to create and evolve integrated, multiple-view models using concepts, relations and model-composition principles used in the given domain. It also facilitates systems/software engineering analysis of the models, and provides for the automatic synthesis of applications from the models. The MIC approach has been successfully applied in many areas, including automotive manufacturing [4], wireless sensor networks [8], and integrated simulation [5], among many others.

A core tool in MIC is the Generic Modeling Environment (GME), which stemmed from earlier research on domain-specific visual programming environments. GME is a domain-specific modeling environment that can be configured and adapted from meta-level paradigm specifications. Thus, based on the paradigm, GME can be adapted quickly to a domain-specific tool that represents a particular engineering domain.

Complex modeling tasks often require knowledge and expertise in numerous scientific and engineering disciplines. The successful use of an environment like GME requires the collaboration and the skillful execution of three different roles: domain expert, environment developer, and experienced programmer. The participants in these roles must synergistically come together in the following way:

Domain Expert. The role of the domain expert is to construct the domain-specific model. They do not need intricate knowledge of GME, rather they only need a basic familiarity that allows them to navigate during model creation. Domain experts do, however, require detailed insight into the various minutiae of the underlying domain.

Environment Designer. The creation of the domain-specific metamodel, which represents the description of a particular modeling language, is a difficult task. The metamodel must contain all of the concepts that the domain expert needs to create a model. The individuals filling this role must have an understanding of both the specific domain, as well as an appreciation of the wide range of GME features.

Component Developers. GME endeavors usually involve software component design also. Components are typically developed to interface GME with different model analysis or simulation tools or to generate code, configuration files or whatever else the given domain needs from the models. The developers must be familiar with both the metamodel and hence, the kind of models that can be built using the given modeling environment.

2. PURPOSE OF THIS PAPER

In this paper, we present a simple case study to introduce the reader to the typical design flow developing a GME-based toolset. The domain of Interactive TV Applications (ITVA) can be described as follows. Digital television allows interactive content to accompany standard broadcasts. However, the development of custom designed interactive content is expensive, and thus, we want to develop a system that will allow the non-technical producers of television programs to build interactive content from a set of high-level building blocks. In effect, we will create a modeling language (i.e., a metamodel), some example models, and a generator that will translate these models into a human and machine readable XML description of the application.

One of the goals of the system is to make it as easy as possible for non-technical producers to build applications to accompany their programs. The applications they build will sit on the right hand side of the television screen and will display one of the following pieces of content:

- A page of text, to be used for items such as news stories, background information, etc.
- A multiple choice vote, for example, "Man of the match," in a football game.
- A menu that allows the user to navigate the interactive content.

The basic on-screen layout and navigational structure of the application has been defined by the user experience department, and producers are not able to change it.

The following use cases were considered:

1) A producer would like to build an application for the World Cup Finals, in which teams are listed and users are allowed to vote for the team they believe will win.

2) The teams are as mentioned in use case 1, but should be listed by groups instead of individually. It should be possible to define the content for teams first and then easily associate teams to groups, where this Group association step should be as convenient as possible.

3) A producer would like to post a pre-game analysis of a match on a page of text. Journalists can then edit and update the page throughout the match. The user interface used by the journalists should not allow them to change the structure of the whole application, rather edit the text only.

4) A producer decides that the wording of a text page was better before the most recent set of changes and would like to revert to the previous version.

3. METAMODELING

The first thing one must do using GME is define a sketch of a metamodel, which is basically a UML Class Diagram extended with some additional concepts. These additional concepts include defining any necessary OCL constraints and also some GME specific features such as configurable model visualization properties. After the metamodel is initially defined, it can be iteratively refined until it reaches a mature state that captures all pertinent features of the domain. This refinement results in an improvement to the domain-specific modeling language (DSML). As the quality of the DSML improves, one can express better domain models using the DSML.

GME metamodels must be created using the MetaGME paradigm, which is installed and registered with GME. MetaGME is just another modeling language; however, its own metamodel can be considered the meta-metamodel. That is, it defines the concepts that are built-in to GME:

- *Atom* – used to represent an atomic element,
- *Model* – used to represent a container element,
- *Reference* – used as a pointer to other elements,
- *Set* – used to group elements, and
- *Connection* – used to associate elements.

These elements are called *first class objects* (FCOs) in GME. FCOs can contain both textual Attributes (of Enumeration, Boolean and Field type) and Constraints, which are OCL-based expressions for providing verifiability for the models. Another important concept in GME is the *Aspect*. Models can have multiple aspects (or viewpoints) that select a subset of the modeling concepts to show to the modeler at once. For example, a model of a distributed software system might have a data flow and control flow aspect.

3.1 First steps in metamodeling

When metamodels are initially designed, the specific domain must be analyzed to find the basic concepts that the metamodel must contain. This technique is similar to UML Class Diagram decomposition in classical software engineering. The nouns in system description documents can often be identified with classes, and the verbs are often represented as operations on these classes. In GME, the metamodels identify FCOs, that is, models, connections etc. In our ITVA domain, the first term that one may identify is *Page*. A Page represents a screenshot taken at any time while the ITVA is running. It contains some form of text or graphics along with possible user operations. We will identify these user operations simply as *Operations*, which will represent the actions a user can take while using an ITVA. Operations may include navigational commands by pushing the color coded keys (e.g., Red, Green, Yellow and Blue) on the remote control or scrolling through a context-aware (i.e., page-specific) menu and selecting one of them. The text that is associated with a page (e.g., "Please vote for the player of the match!") will be defined by a string attribute of the page.

3.2 State machine-like metamodel

ITVAs having Pages and Operations can be thought of as State Machines, where Pages are the states (which we can represent using Atoms in GME), and the Operations (different user actions) are transitions between states (which we can represent using Connections). Given an active page (i.e., a source state) and a user action (e.g., pushing a button), the system must be able to find the next page where that user action leads (that is, the destination state) unambiguously. Thus, we can represent our system as a deterministic state machine. We create a container class named *App* (which stands for Application) to hold both Pages and Operations (Figure 1).

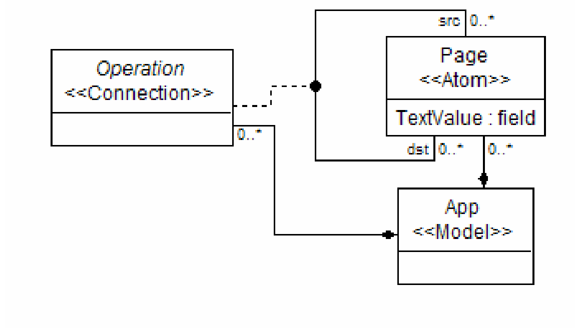


Figure 1: State Machine-like Metamodel

We define five distinct types of operation (transition): four "colored" connections, representing the colored keys on a typical remote controller, and one connection that will represent a menu item selection. These are all derived from the abstract Operation element. Notice the curved arrow and the UML stereotype "ConnectionProxy" in Figure 2. Metamodels can be organized into separate containers called *ParadigmSheets* in MetaGME. Two objects in separate sheets can represent the same concept (e.g. Operation). Instead of just using the unique names, MetaGME applies references and calls them proxies to make this relation. The user can just click on a proxy to navigate to the original definition of the given concept.

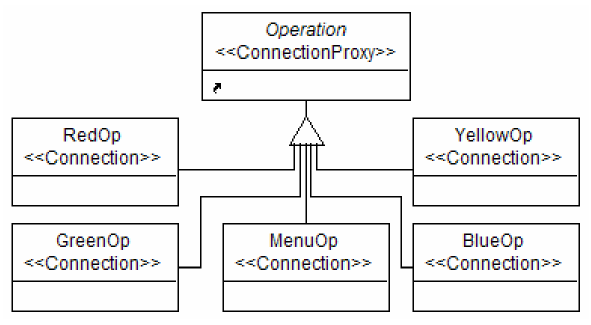


Figure 2: Operation Specialization

4. BUILDING A MODEL

A metamodel must first be interpreted with the MetaGME interpreter, and the generated paradigm description (an XML file which has an "xmp" extension) must be registered in the GME registry before any models of that paradigm can be built. After doing both of these steps, we can create a model of an ITVA that will allow users to vote for the player of the match after a football game. The first step in building this application is to create a new project in GME whose metamodel is our ITVA paradigm, and then to insert an App element into the Root Folder (the top level container) of this project.

4.1 A simple model of a voting application

We continue creating our, "Man of the Match" application by inserting all Pages inside one App element and then connecting the Pages with the possible Operations as shown in Figure 3. One benefit of this simple architecture is that users can view and comprehend models simply by opening the application model (App) and viewing all pages and transitions (the colored connections) among these pages at once. For the voting application, we have placed 5 pages inside an App, where the initial page contains the invitation to vote, and the 4 choices are connected to the initial page through colored connections, which correspond to the typical buttons on a remote control. Although the players and the start page are represented by different icons, they are of the same kind (Page). This is possible because GME allows the icon that represents an element to be replaced at any time.

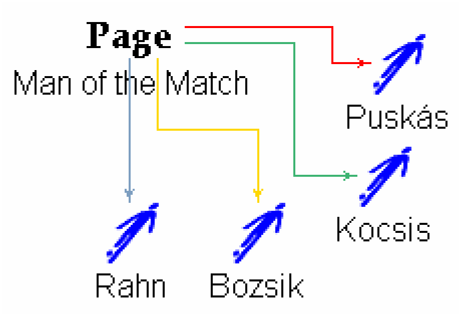


Figure 3: 'Man of the Match' Vote

Because we did not specify any sort of hierarchical containment in our metamodel, no other model can be opened to view all of the pages in an App; thus, exploring and navigating such a model is trivial. However, as the number of Pages increases, deciphering transitions between the pages becomes much more difficult.

4.2 A model of the World Cup

If we want to build a model of the World Cup Finals, we could have a 'Start' page and four sub-pages (Referees, Teams, Groups and Statistics) accessible through color coded operations. The pages for the teams would list all 16 participant teams as menu items. The Groups page would feature 4 group symbols (A, B, C, D) as color coded operations. Because Groups are Team associations, each Group page would display the four teams associated with that Group. Here again we would use menus to represent the team choices on the Group pages, which is the reason that the connections are in black.

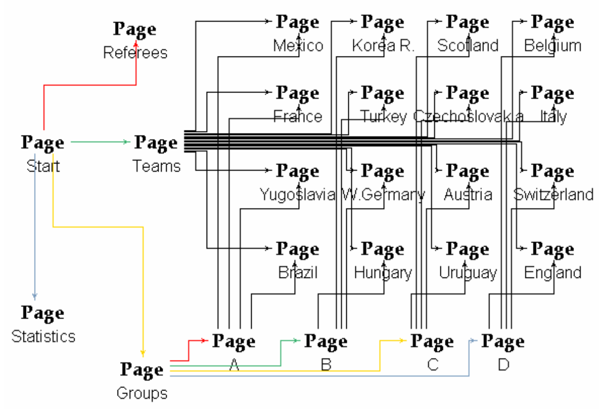


Figure 4: World Cup Final 1954 model

As shown in Figure 4 above, larger models created with this metamodel are not very user friendly; the large number of connections makes the model confusing and difficult to comprehend. Consequently, we can conclude that this architecture is not scalable.

5. METAMODEL REFINEMENT

To overcome the flat structure and cluttered connections seen in the previous section, we can design a metamodel which allows hierarchical model construction. One drawback of the previous design was that it relied on connections in order to specify the transitions between pages. Connections in GME are relatively strictly defined, considering that the connectible elements must be present in the same container as the connection in any of the following 3 ways:

1. The connectible element is physically in the same container as the connection.
2. The connectible element is a child exposed as a port of a model that is in the very same container as the connection.
3. The connectible element is exposed as a port of a model which is accessible through references from the connection's container (the connectible element is a *referenceport*).

Defining connections in a metamodel with hierarchy requires attention to the rules mentioned above. Had we decided to introduce hierarchy to avoid the cluttered models, we would have faced these limitations of connections, specifically the need to expose the connection ends as ports of their parent. We will define another metamodel with the following considerations: in order to express relations among Pages without using connections, we will select the Containment relationship to have a special meaning: a Page will contain only those Pages which can be accessed through a user action such as pushing a colored button on the remote. The Operation which leads from a source page to a destination page will be specified as an attribute of the destination page, with an Enumeration value of either Red, Green, Yellow, Blue or Menu values. The page contained in the Root Folder (the default top level container in GME) will be considered as the start page of an application, and any subpages will be contained by this start page.

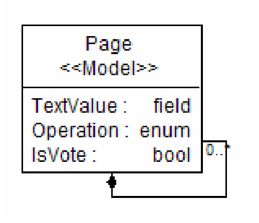


Figure 5: Class Diagram of the New Metamodel

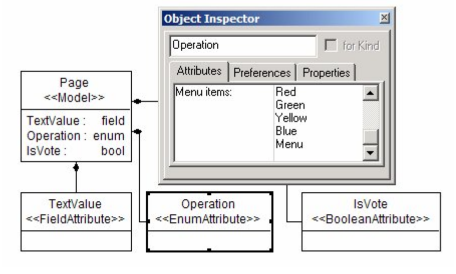


Figure 6: Operation Attribute of Page

In this manner, we have created a metamodel that allows for creating much more structured models, in which every subpage is associated with the page from which it is accessible.

5.1 A simple model for voting

Using the paradigm outlined above, we can build a voting application by first creating a top level Page in a new project. This start Page will function as the introductory page which welcomes the user into the ITVA and sets the plot. In our voting model, this start page displays the invitation to the vote: "Who do you think was the player of this match?" There will be four players from which to choose, and according to our paradigm, all of these options must be pages themselves. More precisely, they are subpages of the voting page. Figure 7 shows four such pages, named for the players one can vote for as the best player of the match.

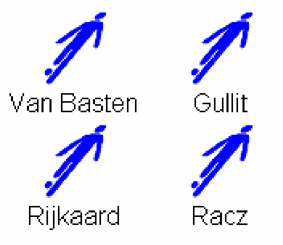
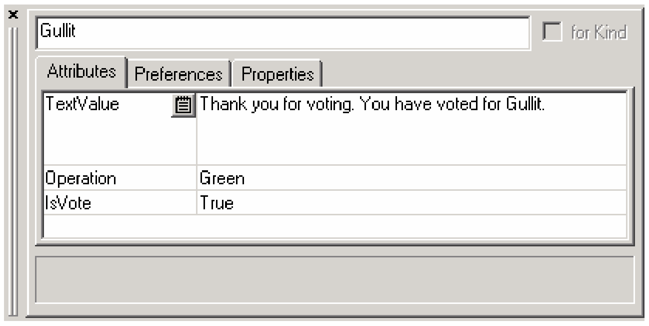


Figure 7: Vote Page for Euro'88

We also need to set the operation attribute of each page so that each possible user input (e.g., pressing the "Green" button on the remote control) results in only one vote. The text value of these subpages will be: "Thank you for voting. You have voted for Gullit." in the event that the user votes for Gullit.



Gullit

for Kind

Attributes | Preferences | Properties

TextValue	Thank you for voting. You have voted for Gullit.
Operation	Green
IsVote	True

Figure 8: Settings for Gullit's page

As far as our voting application model is concerned, we are done: there are 5 total pages, and one of them is the start page. However, as we examine our model, we realize that although the architecture is simple, the page that will be displayed as a result of the user's input is not obvious, because all subpages which represent votes look the same: blue icons of football players ready to kick the ball. The only way in which they differ is their attributes, but these may not be visible enough for a non-technical producer who develops the application, and who might forget to set the Operation attribute of each Page to a distinct value, so that no two pages have the same value for their operation attribute. One solution is to define a simple OCL constraint to ensure the uniqueness of the attribute values. Moreover, a GME Decorator component should also be used, since it may even be more intuitive to a non-technical user. The OCL constraint which follows ensures that any Page contains at most one inner Page with the 'Red' attribute:

```
self.parts(Page)->select( b: Page |
        b.Operation = Red)->size <= 1
```

The validity of OCL expressions can be checked on different user events; the one we will select for this example is the "On Close Model" event. This means that whenever a user attempts to close a Page, this expression will be evaluated. If the children of the to-be-closed model are not properly specified, a dialog will be shown with details about the violation. In the event that the constraint is violated, the metamodel designer must decide whether or not the modeler should be allowed to proceed without correcting the error; to ensure that the error is corrected a priority of '1' needs to be set for the constraint.

5.2 Creating a Decorator

We want to build a decorator that will reflect the Operation attribute's value while showing the subpage. One obvious choice is painting the subpage in the same color as value of the Operation attribute (i.e., if the Operation attribute of a subpage is "Red," then display that page in the color red). GME's modular architecture facilitates one to plug in a component that helps the GME graphical user interface draw an element. In order to do this, an MS COM component must be created which implements the IMgaDecorator interface defined in the MgaDecorator.idl file in the Interfaces directory of the GME installation. In the case of our simple paradigm, two methods need to be implemented: Initialize and Draw. In the Initialize method, we get the Operation attribute's value, and store it in a member variable. In the Draw method, we will simply draw a filled ellipse using the color stored in our member variable. If the Operation attribute is not a color but instead is a Menu, then we will draw a rectangle, which will resemble a menu item. If the user changes the attribute value of a page, the user interface will be updated accordingly, so that the user will see a much more intuitive view of the possible navigational options. Furthermore, if a page has its 'IsVote' boolean attribute set to True, the buttons will show additionally a 'tick' mark. The users may appreciate this visual aid.

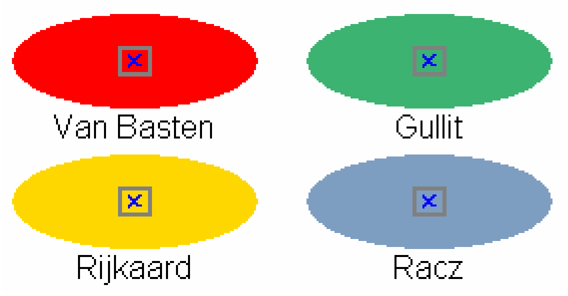


Figure 9: Decorated Voting Page

Setting up a decorator can be easily done at run-time by inserting the decorator component's progid (e.g., "Mga.Decorator.MyDecorator") into the to-be decorated FCO's preference field called, "Decorator". In this manner, one can test a decorator while developing it for one element. After we are pleased with its functionality, we can set the default decorator for certain FCO kinds in the metamodel itself. In our case, we can set it for the 'Page' FCOs by specifying the decorator's progid in their decorator attribute. After setting a decorator for certain kinds in the metamodel, a project can be opened and viewed correctly only if the decorator is registered as a Windows DLL. This means that if we decide to ship our paradigm and model to another computer, then we must also ship the decorator.

5.3 Further refinement of the metamodel

Because the containment of subpages means that one can transition between the two pages, it is hard to imagine how page duplication could be avoided in our model in cases when a page is accessible from more than one page. For instance, in our World Cup scenario, consider an application in which each team can be accessed from either the Team Listings page or from its group's page. The Team Listings page may list the teams alphabetically, while the Groups page may want to list the teams by their international ranking. In such a case, our current paradigm forces us to have two pages for each team: one defined as a subpage on the Team Listings page, and another defined as a subpage in its group page. We would like to avoid this duplication, which might lead not only to inconsistency, but extra effort whenever a team is updated. To alleviate this problem, we will define *Page References* (PageRef for short) in the metamodel. These Page References will allow us to insert the same page into multiple container pages, much like a pointer variable in C++, so that whenever the original Page is updated, all references to that Page point to the updated model and hence, do not need to be modified. PageRefs are contained by Pages, so the metamodel in Figure 10 shows both the reference relationship between them (arrow) and the containment (diamond). In general, however, the reference and containment relationships are independent.

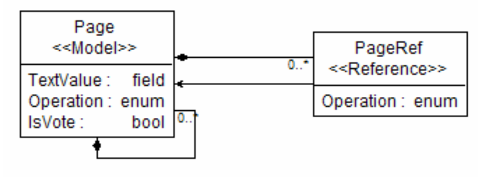


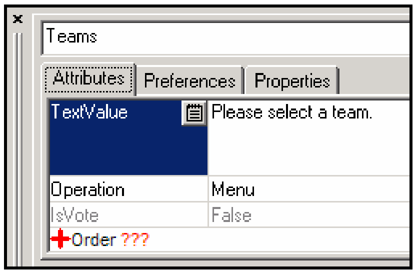
Figure 10: Metamodel with References

5.4 Building a Model for the World Cup

The World Cup Finals application can be modeled by listing the participant teams, the referees, the grouping of the teams, and the World Cup history. Considering that the number of teams is usually more than 20, instead of using the color coded operations described above (which allow only 4 options at one time), we could instead create a menu based application (because the number of menu items that can be displayed at one time is not limited). The start page could contain the following categories: Teams, Referees, Groups, History and Voting. The text associated would be: "Welcome to the World Cup Interactive TV Application. Please select one option."

Figure 11: Top Menu Items on Main Page

The Operation attribute allows us to specify these options as menu items, but does not allow us to specify the ordering of these menu items. This requires some action: we either specify an ordering mechanism, or we must introduce into the metamodel the capability to specify menu item ordering. The latter could be done by adding an integer attribute to each subpage, which will be set by modelers to specify ordering: smaller values indicate higher positions in the menu.



TextValue	Please select a team.
Operation	Menu
IsVote	False
+Order ???	

Figure 12: Metamodel with an Order Attribute Added

In the case of color coded operations, we would find this Order attribute redundant because the color coded operation should always follow the remote controller's standard layout: Red, Green, Yellow and Blue. For this reason, we choose not to alter the metamodel. Instead, we consider each menu item's position in the model to be the order in which it is displayed on the television screen, which we believe is an intuitive ordering algorithm. Thus, all subpages which have the menu item as their invoking operation will be sorted based on their position (X, Y) in their parent page. This step can be carried out by the model interpreter that generates the XML file specifying the ITVA.

5.4.1 *Grouping of the teams*

One of the use cases requires that the grouping of teams should be easy. Establishing the groups for our World Cup model consists of creating 6 Group subpages, Group A through F, and sorting the teams into these Group pages. The team pages already exist, so we either copy them (as a subpage) into the Group pages, or we create a reference to them by using the PageRef construct. Copying a team page creates redundancy, which in turn could increase the effort needed to update information about the team (e.g., total yellow cards given) and might easily lead to inconsistencies. For this reason, we choose to use references. A reference of type PageRef can point to any Page, and Figure 13 shows the GME model browser with a sample application in which we have inserted a PageRef into Group F which points to Brazil's team page.

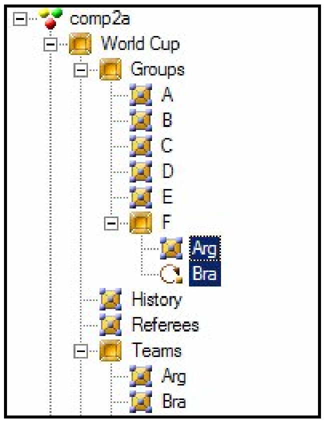


Figure 13: Argentine Team Duplicated, Brazil Team Referred

As we build a few separate Team pages, we see some reoccurring patterns. For example, each team page must contain certain subpages, such as History, Coach, Squad and Yellow Cards. These subpages must be created inside every team's page, and for consistency, we would like them all to organize their content in the same way. This implies that our job would be eased if some common "template", which contains the elements common to all team pages, could be created first, and then each team could specialize this template with their team-specific information. For example, the History subpage of Brazil (e.g., "Brazil won the world cup 5 times") would differ from Germany's History subpage (e.g., "Germany won the world cup 3 times"). The Squad (i.e., the players on a team) subpages also typically contain 22 players, so we could also create a template for Squad pages.

5.5 Prototyping

GME provides a mechanism similar to type inheritance in Object Oriented Design, which allows one to create such common types during modeling and later create subtypes and/or instances of these types in the same model. Every model, atom, set or reference in a model can be subtyped or instantiated. Looking at our example, we could create a History page-type, or a Squad page-type containing 22 subpages for players. However, we can also move one step up in the hierarchy, and create a Team page-type, which would contain the Squad page and History page. Then whenever we want to add a new team, we simply reuse the created team type by creating a new instance or subtype of our base type. Subtypes are much more flexible than instances, because they allow new children to be added, while instances are read-only derivations of their type as far as their internal structure is concerned. In case of attributes there is no such limitation: attributes of instances can be modified just like those of subtypes'. In the world cup scenario, we would create an Archetype Team page with all its internal subpages included already, and derive our specific subtype team pages (e.g., Argentina, Brazil, etc.) from this Archetype Team page.

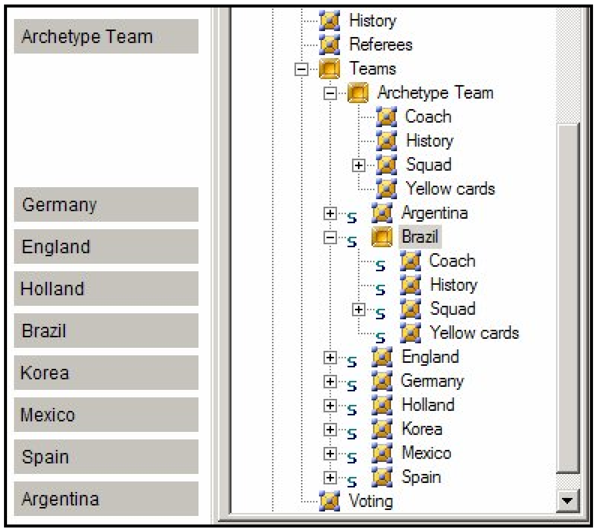


Figure 14: Archetype Team and its Subtypes

This mechanism can be called prototyping, and it has a consequence on the model: the prototype pages are also part of the model, which means in our case that the Archetype Team page will become a regular menu item on the Teams page (Figure 14). Because the Archetype Team does not represent any specific team, we wish to avoid this. To avoid mixing prototype pages with subtype pages, we must somehow distinguish between the two.

5.6 Metamodel Update

Introducing an additional attribute or enumeration item for Pages in our metamodel could solve our problem. We choose to add an additional Enumeration item, and we add the 'None' value to the Enumeration Attribute called Operation. This will represent inactive content, so that each page marked with this 'None' value as its invoking operation will be considered as non-existent from the application's point of view.

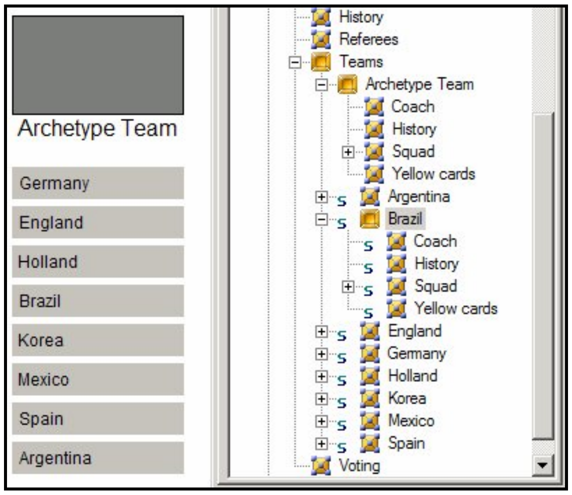


Figure 15: Archetype Team Page Now Inactive

We also update our decorator component to display inactive content differently (Figure 15).

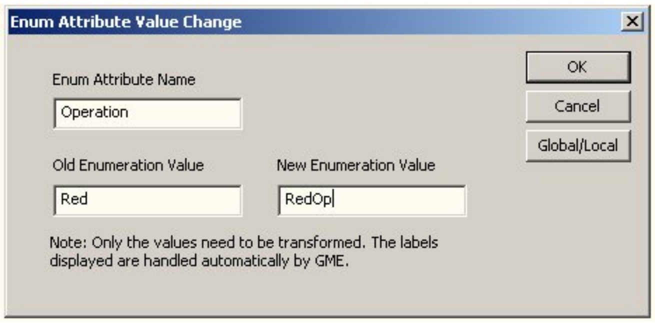
5.7 Model Update

After we update our metamodel, we need to reinterpret it and register the new version of the paradigm. During the registration of the paradigm's new version, the old version is backed-up into a file created by appending the paradigm version's unique id to the filename. If this file remains untouched, GME will allow users to open models created with older version of the paradigm. However an Upgrade option is also available for users. This upgrade can be binary or XML based. In certain cases (like EnumAttribute values modification or renaming) both will fail, which forces users to do model transformations. Possibilities for model migration include using a GReAT transformation, or using the ModelMigrate tool, which is suitable for simple XSL-based model transformations on the XML representation of the models.

5.7.1 Migrating Models

The ModelMigrate.exe standalone executable tool allows users to define model transformations as distinct search and replace operations, and then to apply these transformations to their models that are in .xml format. The result of applying such a transformation is a new .xml file. All operations consist of applying XSL Transformations, which have been developed by the GME developers using knowledge about the GME XML file format (.xme) and its schema as defined in the mga.dtd file. The users do not need any knowledge of XSL transformations; rather, they have to use only the simple and intuitive 'Rule Editor' dialog in ModelMigrate's GUI to generate the needed XSL Transformation Scripts by filling in search and replace style dialogs. The following transformational rules are included:

- KindNameChange: an FCO kind name has been renamed in the metamodel,
- AttrNameChange: an attribute name has been renamed,
- AttrTypeChange: an attribute's type has been altered,
- EnumAttributeValueChange: an EnumAttribute's item has been renamed,
- RemoveKind: removing an obsolete kind,
- RemoveAttr: removing an obsolete attribute,
- Atom2Model: an FCO defined earlier as an Atom has been changed to be a Model, and
- Model2Atom: an FCO defined earlier as a Model has been changed to Atom.


Enum Attribute Value Change

Enum Attribute Name: Operation

Old Enumeration Value: Red

New Enumeration Value: RedOp

Note: Only the values need to be transformed. The labels displayed are handled automatically by GME.

Figure 16: Search & Replace-Style Dialog

The rules specified in ModelMigrate.exe can be generated into separate .XSL files or into one combined file. The former allows for more control over transformations because the rule application order is clearly defined by the order in which the user executes the XSLT scripts.

For ITVA, we can imagine a scenario in which the distinct values of the Operation enumeration attribute of Pages need to be renamed in the metamodel, e.g., "Red" renamed to "RedOp", "Green to GreenOp", etc.). This kind of change currently breaks the GME Upgrade through XML feature, so models created earlier cannot be automatically upgraded. In this situation, we could use the ModelMigrate tool to define 4 rules, each of type EnumAttributeValueChange. Each will search for the occurrence of a different attribute Operation and rename it appropriately.

5.7.2 The GReAT Framework

As mentioned above, making changes to a metamodel can invalidate existing models. When these changes are small, then a simple tool such as ModelMigrate.exe can be used to migrate the old models, so that they conform to the new version of the metamodel. However, when these changes are more significant, a more powerful tool is needed to perform the transformation. For example, consider a modification to a metamodel in which we create two new derived types of an existing connection, and we make the existing connection class abstract. If we need to migrate the existing connections (which are now no longer valid because the base connection class was made abstract) based on the attribute values of the connected elements, the problem is no longer a simple text find/replace issue, and consequently, ModelMigrate.exe will not suffice.

GReAT, the Graph Rewriting and Transformation Language [6], is a powerful model transformation framework that can be used in such situations. GReAT builds upon the formalisms of graph grammars by considering model transformations as graph transformations in which the input model is the source graph and the output model is the target graph. GReAT uses the UDM Framework [7] as its underlying data model, which provides GReAT with programmatic C++ access to the input and output models. It is important to note that both GReAT and UDM are fully-developed frameworks for specifying complex model interpreters, similar to BON2: GReAT is a visual, domain-specific method, and UDM is a C++ (textual) domain-specific method.

GReAT is a visual language, completely integrated within GME, that uses elements from the metamodels of the input and output models to define transformation rules that are sequentially applied to the input model. In the terminology of graph grammars, each transformation rule consists of a left-hand side (LHS) and a right-hand side (RHS). Roughly speaking, in each transformation rule the user defines a pattern to be matched (the LHS) in the host graph and an action to apply (the RHS) whenever an occurrence of the LHS is matched. These actions can include creating new objects and associations, deleting old objects and associations, and modifying attributes.

A GReAT transformation is itself specified as a model in GME; that is, GReAT contains a metamodel which allows one to specify model transformations. This ability to completely specify model transformations as models themselves shows the level of expressiveness found in GME.

The first step in writing a GReAT transformation is to create a new project in GME of type “UMLModelTransformer”. Next, the metamodels of all input models and all output models are attached to this project. One nice feature of GReAT is that any number of models and metamodels can be used during a transformation. For example, one can have an input model, an intermediate model, and an output model all used during the same transformation. In the case of our model migration example of section 5.7, we would attach both the old version of our metamodel and the new version of our metamodel to our project. After specifying required configuration information such as the names and locations of the input and ouput models, we define our transformation rules. In our example, an EnumAttribute value of our “Page” model has been renamed, so our transformation rule is a pattern that finds the EnumAttribute in the old model and creates in the updated model a “Page” with its EnumAttribute value set to the new name.

We should note that GReAT is much more than a model migration tool; it can be used to perform complex model transformations between models of very different heterogeneous domains. For such a usage of GReAT, please see [9], in which GReAT was used to specify a C-Code generator for Simulink models.

5.8 Scripting and Automation

Building models sometimes consists of doing repetitive tasks. As we have seen in our World Cup Application model, each team may have its 22 players listed on their Squad page. Inserting 22 player pages and setting the attributes of each can become very tedious and tiresome, especially compared to creating one player page, setting its attributes, and then executing a script written in one’s favorite scripting language that duplicates this object 21 times. Several scripting engines which implement the ActiveScripting interface can be used inside GME. For instance, the Python and Perl distributions by ActiveState have such components, and JScript and VBScript are preinstalled on every Windows XP system by default. The preferred scripting engine can be selected anytime in the File/Settings menu of GME. Assuming that we are currently viewing the Squad subpage of the Archetype Team Page (created in Section 5.5), the following Python code will create 21 players by duplicating an existing player named ‘1’.

```
# cloning 21 times an element called ‘1’
# variable ‘it’ represents the viewed model
> for i in range(2,23):it.Duplicate('1', str(i))
```

Repetitive tasks can be automated easily, or complex model building user functions can be written by using commands defined in the IGMEOLEIt automation interface (found in the Interfaces/gme.idl file of the GME distribution).

5.9 Model Libraries

Libraries are a potential candidate for giving users read-only access to pre-built models or model fragments in an integrated fashion. Libraries are read-only copies of a project contained in another host project (these two projects must share a common paradigm). Library constructs are either:

- copied into the project: in this case the copy is a stand alone structure freshly created, and the copy loses its ties with the original, and there is no write protection of any kind (the copy can be freely modified as desired). Copying somewhat defeats the purpose of libraries and hence, it is not recommended.
- subtyped or instaniated: instantiation allows only attributes to be altered, while subtyping allows both the addition of new children and the altering of attributes.
- referenced, if such a referring kind has been defined in the metamodel.

When libraries evolve and need to be updated in a hosting project, they can be refreshed, so that relationships built within the hosting project to the library will be preserved. This capability only refers to the subtype and reference relationships, because as mentioned previously, an object copied from a library into the host project has no dependency information whatsoever with the original element.

6. USE CASE DISCUSSION

We have shown how a model can be built for the World Cup Finals (Use Case 1) and how teams can be easily associated to groups (Use Case 2). In the following sections, we will discuss role based permissions for updating pages (Use Case 3) and a version control mechanism that will allow us to revert to previous versions of pages (Use Case 4).

6.1 Use Case 3

6.1.1 Scenario 1

A producer could create an archetype page providing the analysis of the football match in the page’s TextValue attribute. An instance of this page can be created, which will be suitable for the Journalist who needs to update the TextValue attribute field containing the analysis. Attributes of this page instance can be changed, but the structure of the page cannot.

6.1.2 Scenario 2

If additional protection is needed, then the whole project could be made read-only, with the exception of the analysis page which needs to be updated by the journalist. There are two ways GME could provide this read-only accessibility:

- By using an add-on, which prevents modifications to the project by the Journalist using a role based security policy. Upon the first modification, this user written add-on could ask for a login name and password, and would abort any transaction in which the journalist tried to modify a page for which they did not have permission.
- By applying the read-only flag to the restricted parts of the project through the Tree Browser's Access context menu item, and allowing only the Producer to remove this read-only flag.

6.1.3 Scenario 3

Journalists could use a specific interface created for them as a model interpreter, which is another type of user-written component. This interpreter would display only a limited GUI, allowing Journalists to see and edit only certain attributes of the pages.

6.2 Use Case 4

GME offers the following options to achieve an undo mechanism:

- Using the multi-user project type, which separates projects into multiple XML files, and supports Check-In and Check Out operations with Rational ClearCase and MS Visual SourceSafe versioning systems.
- Generic built-in Undo/Redo mechanism. The size of the Undo queue can be set in the File/Settings menu anywhere in the range of 1 through 99 operations.
- Preserving old pages inside the model, but marking them as inactive. This is a proactive approach taken by the user.
- Using the type inheritance feature. Instead of modifying the text, the user can create a new instance or subtype of the latest version of the given model and modify the text attribute of this new model. This way the whole history is preserved and can be conveniently navigated.
- Writing a user component (an interpreter) that could be used to update text and automatically save previous values to permanent storage whenever changes are committed. If a previous wording is desired later, then user written component could reload the earlier values from the permanent storage.

7. ACCESSING THE MODEL PROGRAMATICALLY

GME's modular architecture allows us to write COM components using any technology and language which has COM interoperability. This means we could use C++, Python, Perl, VB6 or any .NET and CLR based language (managed C++, C#, VB.NET) to write a COM based component. GME provides a component skeleton generator called CreateNewComponent.exe (found in the SDK directory), which makes the creation of new C++ components trivial: the user needs only to specify the component's name and the related paradigm name, and a MS Visual Studio solution and project file, along with an empty skeleton, are set up. Three different kinds of components, offering different levels of abstraction, can be created with the CreateNewComponent tool:

1. Raw COM based C++ components are at a very low-level, and thus offer the fastest method of accessing models. The COM interfaces that need to be used are defined in the Interfaces subdirectory (in particular mga.idl) of the GME installation. Basic MS COM knowledge is required to write such components.
2. BON (Builder Object Network) C++ components use a higher level interface to access models, and rely on Microsoft VC6-style containers. No MS COM knowledge is required to use BON.
3. BON2 C++ components are Standard Template Library (STL) based, and provide additional domain-specific interface generation capabilities.

Reusable component skeletons have also been published for both Python and C#.

7.1 BON2 Interface

The BON2 object network is a C++ class network, which is created in correspondence to the model upon which the interpreter is invoked. For instance, in the case that a project has a model with two atoms inside, then the BON2 initialization code will create three C++ objects: one instance of a BON::Model class and two instances of BON::Atom, and these objects will compose an object network, by maintaining their parent-child relationship. For instance, the BON::Model object will have methods to return its BON::Atom children, and the BON::Atom objects will each have a method to return the BON::Model object in which they are contained.

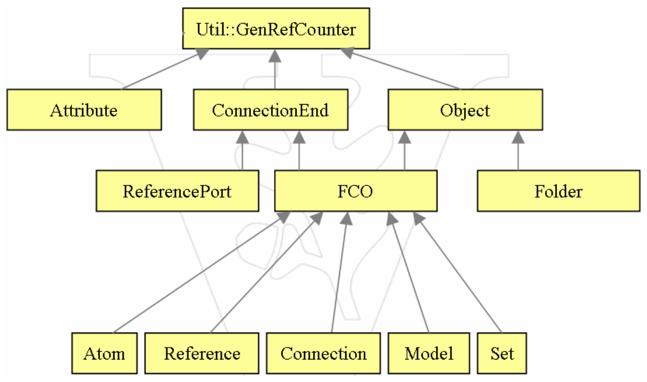


Figure 17: BON2 Class Hierarchy

These classes are wrappers for their inner BON::ModelImpl and BON::AtomImpl pointers, which do the actual implementation. The 'Impl' class hierarchy is similar to the wrapper class hierarchy shown in Figure 17.

BON2 Components execute slower than Raw COM Components, because these C++ classes and the relationships among them must be created around the COM pointers which GME exposes.

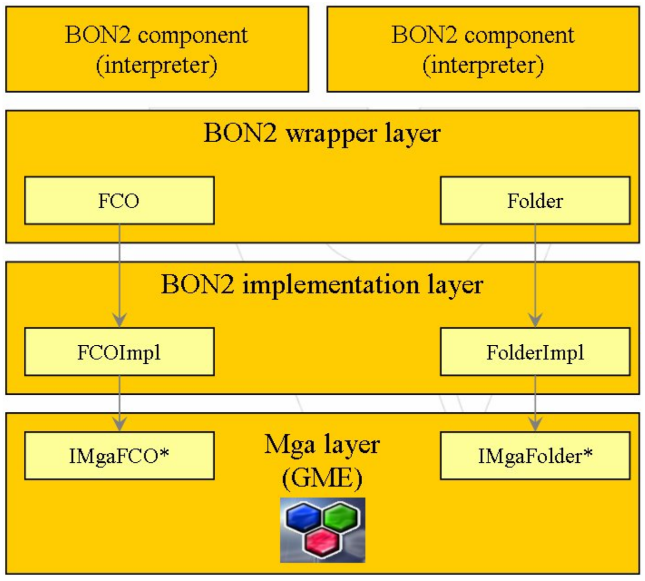

BON2 component (interpreter)
BON2 component (interpreter)
BON2 wrapper layer
FCO
Folder
BON2 implementation layer
FCOImpl
FolderImpl
IMgaFCO*
Mga layer (GME)
IMgaFolder*

Figure 18: BON2 Layers and the Mga Layer of GME

BON2 interpreters can take advantage of C++ method calls to BON2 classes such as ModelImpl and ConnectionImpl (declared in SDK/BON/BonImpl.h file), which have much nicer public interfaces than the IMgaModel and IMgaConnection (defined in Interfaces/Mga.idl file).

```
class ModelImpl : public FCOImpl
{
// methods of BON::ModelImpl
// . . .
std::set<FCO> getChildFCOs
    ( const std::string& strFCO
    , const MON::Aspect& aspect = MON::Aspect() );

std::set<Atom> getChildAtoms
    ( const MON::Aspect& aspect = MON::Aspect() );

std::set<Model> getChildModels
    ( const MON::Aspect& aspect = MON::Aspect() );

std::set<Set> getChildSets
    ( const MON::Aspect& aspect = MON::Aspect() );

std::set<Reference> getChildReferences
    ( const MON::Aspect& aspect = MON::Aspect() );

std::set<Connection> getChildConnections
    ( const MON::Aspect& aspect = MON::Aspect() );
};
```

```
class ConnectionImpl : public FCOImpl {
// methods of BON::ConnectionImpl
// . . .
std::multiset<ConnectionEnd> getConnEnds
    ( const std::string& strFCO );

ConnectionEnd getConnEnd
   ( const std::string& strRole
   , const std::string& strFCO );

ConnectionEnd getSrc( const std::string& strFCO );
ConnectionEnd getDst( const std::string& strFCO );
};
```

7.1.1 Generating a Domain-Specific BON2 Interface

Just as a metamodel defines a Domain Specific Modeling Language, it can also be considered as definition of a Domain Specific Interface. This domain specific interface can be generated by the BonX interpreter, which comes with GME and is executed on the metamodel. The BonX interpreter generates a header and source file containing C++ class definitions for all elements defined in the DSML. These classes have methods specific to their corresponding elements in the DSML. For example, if an element in a DSML has a bool attribute named "IsVote", then the generated C++ class for that element will have getter and setter methods: isIsVote() and setIsVote(). If a container element may contain Page and PageRef elements, then it will have getter methods called getPage() and getPageRef() to return its Page and PageRef elements, respectively. For example, the Page class associated with the model Page (defined in our metamodel) gets a specific interface as follows:

```
class PageImpl :
       virtual public BON::ModelImpl {
public:
typedef enum
{
       None_Operation_Type,
       Red_Operation_Type,
       Green_Operation_Type,
       Yellow_Operation_Type,
       Blue_Operation_Type,
       Menu_Operation_Type
} Operation_Type;

// attribute getters and setters
Operation_Type  getOperation();
std::string getTextValue();
bool isIsVote();

void setIsVote( bool val);
void setOperation( Operation_Type val);
void setTextValue( const std::string& val);

// kind and role getters
std::set<Page> getPage();
std::set<PageRef> getPageRef();
}
```

Authoring a component using this domain-specific BON2 interface is easier and less error prone than using the generic BON2 interface, as illustrated by the following three code snippets, all three attempting to obtain the "TextValue" attribute of a Page; the first uses the generic BON2 API, the second has a typo error, thus will throw an exception, and the third uses the domain-specific BON2 interface.

```
std::string s;
// using a generic BON2 call
s=p->getAttribute("TextValue")->getStringValue();
// using a mispelled attr name: exception
s=p->getAttribute("TetxValue")->getStringValue();
// using domain specific BON2 code
s=p->getTextValue();
```

7.1.2 Implementation

Our interpreter's task is to produce an XML description of our ITVA models. The interpreter will traverse the model starting from the start page and dump details about the page and all of its possible operations, and then continue with the subpages.

Provided that our Visitor class is derived from an `XmlWriter` class that helps us write XML in a nice fashion using its WriteStartElement, WriteEndElement, WriteAttrString and WriteCData methods, a BON2 component using a domain-specific interface can be implemented as follows:

```
void Visitor::visitPage( Page& page )
{
if( page->getOperation() ==
     Operation_Type::None_Operation_Type )
        return;

WriteStartElement( "page");
WriteAttrString( "name", page->getName());
WriteAttrString("op", page->getOperationStr());

std::set<BON::Reference> refs;
refs = page->getReferredBy();
if( refs.size() >= 1)
{
    WriteAttrString( "id", page->getID());
}

WriteStartElement( "text");
WriteCData( page->getTextValue());
WriteEndElement(); // </text>
std::list<BON::FCO> chld;
chld = sorted( page->getChildFCOs());

std::list<BON::FCO>::const_iterator i, e;
for(i = chld.begin(),e = chld.end(); i != e;++i)
{
        if( Page( *i)) visitPage( *i);
        else if( PageRef( *i)) visitPageRef( *i);
}

WriteEndElement();
}
```

The domain specific API generated by the BonX interpreter is used heavily by the code fragment above. For instance, we have domain specific C++ classes such as Page and PageRef. These classes correspond to the kinds defined in our language. Thus, the getOperation() and getTextValue() methods of the Page class inquire for a specific Operation and TextValue attribute. In the last 'for' loop in the code above, we can see that Page(*i) is an expression that tests if the class pointed by the iterator i (*i) is of kind Page or PageRef, and based on this test, the correct visit method is called. The specific child getters getPage() and getPageRef() are not used here, because we need to sort the combined set of children in order to dump menu items in the correct order. Therefore, it is easier to inquire for all children and then sort them, rather than asking for 2 distinct sets of specific children, taking their union, and sorting them later.

7.2 Raw COM Interpreter in C#

Users can generate empty skeletons of C++ Raw Components with CreateNewComponent.exe, and in such cases the implementation details of a COM server class are hidden from them. This is only half of the story, however, because the users still need to handle, store and pass COM pointers according to the COM rules, which are related mostly to the COM way of reference counting. Higher level languages like C# and VB.Net hide the complexity regarding this reference counting, which is why authoring components in these languages (considering the rich set of class libraries they offer) becomes attractive. The nice .NET-COM interoperability allows any C# programmer (who may be a beginner in COM) to write interpreters for GME. The details of reference counting that dominate C++ based COM interpreters are not relevant in C#. Creating such an interpreter in C# begins by creating a New Visual Studio project in C# of type Class Library. After the project is created, we must add the following type libraries as references: Mga, MgaUtil, Meta, Core and Gme. We then declare a class which will implement the required GME interfaces: IMgaComponentEx, IMgaVesionInfo, IMgaComponent. The class will be annotated by progid and guid attributes, as shown below.

```
[Guid("12345678-9abc-def0-1234-56789abcdef0")]
[ProgId("Mga.Interpreter.CS4Int")]
public class MyMgaComponent : IMgaComponentEx,
IMgaComponent, ImgaVersionInfo
{
 // . . .
}
```

Whenever an interpreter is invoked from GME, the InvokeEx method, defined in IMgaComponentEx interface, is the first method called. The interpreter's main code should be included here:

```
public void InvokeEx
  ( MgaProject prj
  , MgaFCO currentobj
  , MgaFCOs selectedfcos
  , int param)
{
 IMgaTerritory terr;
 transactiontype_enum mod;
 mod=transactiontype_enum.TRANSACTION_GENERAL;
 prj.CreateTerritory( null, out terr, null);
 prj.BeginTransaction( terr, mod);
 try {

  // do the hard work

  prj.CommitTransaction();
 }
 catch(Exception) {
  prj.AbortTransaction();
 }
}
```

7.2.1 *Implementation*

Provided that we have an instance, x, of the `XmlTextWriter` class that helps us write XML in a nice fashion by its WriteStartElement, WriteEndElement, WriteAttributeString and WriteCData methods, a Raw COM C# interpreter can be implemented as follows:

```
void InvokeEx( MgaProject project
             , MgaFCO currentobj
             , MgaFCOs selectedfcos
             , int param)
{
  // entry point of interpreter
  MgaFolder rf = project.RootFolder;
  foreach( MgaFCO child in rf.ChildFCOs)
    visitTopPage( child as MgaModel);
}

void visitTopPage( MgaModel modelfco)
{
  x.WriteStartElement( "TVApp");
  x.WriteAttributeString( "name", modelfco.Name);

  foreach( MgaFCO child in
    ⇘ modelfco.GetChildrenOfKind( "Page"))
    visitPage( child);
  x.WriteEndElement();
}

void visitPageRef( MgaReference fco)
{
  x.WriteStartElement( "pageref");
  x.WriteAttributeString( "name", fco.Name);
  x.WriteAttributeString( "op",
    ↳ fco.get_StrAttrByName( "Operation"));

  if( fco.Referred != null)
    x.WriteAttributeString( "ref",
      ↳ fco.Referred.ID);

  x.WriteEndElement();
}

void visitPage( MgaFCO fco)
{
  x.WriteStartElement( "page");
  x.WriteAttributeString( "name", fco.Name);
  x.WriteAttributeString( "op",
    ↳ fco.get_StrAttrByName( "Operation"));

  MgaFCOs rby = fco.ReferencedBy;
  // it is referred, we dump id too
  if( rby.Count >= 1)
    x.WriteAttributeString( "id", fco.ID);

  MgaModel m = fco as MgaModel;
  if( m != null)
  {
    x.WriteStartElement( "text");
    x.WriteCData( fco.get_StrAttrByName(
      ↳ "TextValue"));
    x.WriteEndElement(); // text

    foreach( MgaFCO child in sorted( m.ChildFCOs))
    {
      // if ‘None’, disregard
      if(child.get_IntAttrByName( "Operation")==0)
        continue;

      if( child is MgaModel)
        visitPage( child);
      else if( child is MgaReference)
        visitPageRef( child as MgaReference);
    }
  }
  x.WriteEndElement();
}
```

The code above generates an XML output as follows:

```
<TVApp name="World Cup">
  <page name="Teams" op="Red">
    <text><![CDATA[Please select a team.]]></text>
    <page name="Mex" op="Menu"
      <text><![CDATA[Team Mexico]]></text>
    </page>
    <page name="Kor" op="Menu"
      <text><![CDATA[Team Korea]]></text>
    </page>
    <page name="Bra" op="Menu"
      <text><![CDATA[Team Brazil]]></text>
    </page>
    <page name="Eng" op="Menu"
      <text><![CDATA[Team England]]></text>
    </page>
    <page name="Spa" op="Menu">
      <text><![CDATA[Team Spain]]></text>
    </page>
  </page>
  <page name="Man Of The Match" op="Green">
    <text><![CDATA[Please select one]]></text>
    <page name="Racz" op="Blue">
      <text><![CDATA[Thank you for voting! You
have voted for Racz.]]></text>
    </page>
    <page name="Rijkaard" op="Yellow">
      <text><![CDATA[Thank you for voting! You
have voted for Rijkaard.]]></text>
    </page>
    <page name="Gullit" op="Green">
      <text><![CDATA[Thank you for voting! You
have voted for Gullit.]]></text>
    </page>
    <page name="Van Basten" op="Red">
      <text><![CDATA[Thank you for voting! You
have voted for Van Basten.]]></text>
    </page>
  </page>
</TVApp>
```

8. CONCLUSIONS

We have presented the features and capabilities of the Generic Modeling Environment, GME, through an example from the domain of Interactive Television Applications. We showed the iterative process of how a metamodel for such a domain can be specified and refined, and how instance models of this metamodel can be created for various applications. We also showed that through its modular architecture, GME allows for the easy addition of user-written components for generating useful artifacts from models. Also briefly mentioned were UDM and GReAT, two additional frameworks for specifying model interpreters.

9. FURTHER READING

[1] Ledeczi A., Bakay A., Maroti M., Volgyesi P., Nordstrom G., Sprinkle J., Karsai G.: Composing Domain-Specific Design Environments, *Computer*, pp. 44-51, November, 2001.

[2] Karsai G., Sztipanovits J., Ledeczi A., Bapty T.: Model-Integrated Development of Embedded Software, *Proceedings of the IEEE, Vol. 91, Number 1*, pp. 145-164, January, 2003.

[3] Karsai G., Maroti M., Ledeczi A., Gray J., Sztipanovits J.: Composition and Cloning in Modeling and Meta-Modeling, *IEEE Transactions on Control System Technology, (accepted)*, 2003.

[4] Long E., Misra A., Sztipanovits J.: Increasing Productivity at Saturn, *IEEE Computer Magazine*, August, 1998.

[5] Agrawal A., Bakshi A., Davis J., Eames B., Ledeczi A., Mohanty S., Mathur V., Neema S., Nordstrom G., Prasanna V., Raghavendra, C., Singh M.: MILAN: A Model Based Integrated Simulation Framework for Design of Embedded Systems, *Workshop on Languages, Compilers, and Tools for Embedded Systems* (LCTES), , Snowbird, UT, June, 2001.

[6] Agrawal A., Karsai G., Neema S., Shi F., Vizhanyo A.: The Design of a Language for Model Transformations, *Journal on Software and System Modeling 5(3)*, September, 2006.

[7] Magyari E., Bakay A., Lang A., Paka T., Vizhanyo A., Agrawal A., Karsai G.: UDM: An Infrastructure for Implementing Domain-Specific Modeling Languages, *The 3rd OOPSLA Workshop on Domain-Specific Modeling*, OOPSLA 2003, Anahiem, California, October 26, 2003.

[8] Volgyesi, P., Ledeczi, A.: Component-Based Development of Networked Embedded Applications, *Proc. of 28th Euromicro Conference, Component-Based Software Engineering Track*, pp. 68-73, Dortmund, Germany, September, 2002

[9] Neema, S., Kalmar, Z., Shi, F., Vizhanyo, A., Karsai, G.: A Visually-Specified Code Generator for Simulink/Stateflow, In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (Vl/Hcc'05) - Volume 00* (September 20 - 24, 2005).