Raising the Abstraction of Domain-Specific Model Translator Development

Tamás Vajk
Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Budapest, Hungary
tamas.vajk@aut.bme.hu

Róbert Kereskényi, Tihamér Levendovszky, Ákos Lédeczi
Institute for Software Integrated Systems
Vanderbilt University
Nashville, USA
{*roby, tihamer, akos*}*@isis.vanderbilt.edu*

***Abstract*—Model-based development methodologies are gaining ground as software applications are getting more and more complex while the pressure to decrease time-to-market continually increase. Domain-specific modeling tools that support system analysis, simulation, and automatic code generation can increase productivity. However, most domain-specific model translators are still manually written. This paper presents a technique that automatically generates a domain-specific application programming interface from the same metamodels that are used to define the domain-specific modeling language itself. This facilitates the creation of domain-specific model translators by providing a high-level abstraction hiding all the cumbersome modeling tool-specific implementation details from the developer. The approach is illustrated using the Generic Modeling Environment and the Microsoft .NET C# language.**

***Keywords*-Domain-Specific Modeling, Metamodeling, Translator, Code Generation**

I. Introduction

Model-based approaches to software and systems engineering are proliferating. Several modeling tools are widely used, such as IBM Rational or Borland Together, support application development with the use of the Unified Modeling Language (UML) [1]. These are excellent tools for general purpose software development. Domain-specific modeling languages (DSML) [2] raise the abstraction level higher by supporting languages that are custom designed to a given domain. They work well when domain experts are the developers who are not necessarily software engineers/programmers. Well known examples of this category include Matlab/Simulink and Labviews. Naturally, DSML tools need to support not only the model building process, but domain-specific model translation including code generation as well.

Metamodeling tools, such as the Generic Modeling Environment [3], provide the ability to design a metamodel that defines a DSML. The metamodel defines the rules of its instance models. The metamodel determines which types of objects are allowed during the modeling process, what kind of attributes or relations they can have, etc. The metamodels are then used to automatically configure a domain-specific modeling environment. These tools allow the graphical visualization of both metamodels and models.

Model-based development requires the creation of a model and then the processing of it as well. Language engineering is handled by metamodeling, but there is no general, automated solution for translating the model to necessary artifacts such as code. A typical model translator has three parts: accessing and traversing the models, processing the information and generating the corresponding output. Model access and traversal requires detailed knowledge of the typically low-level generic API of the modeling tool. This is typically the most cumbersome and least intellectually stimulating part of creating a model translator. In this paper, we present a technique that automatically generates a high-level, domain specific, object-oriented API from the metamodels. This API hides the low-level, tool-specific details and provides an intuitive interface for the model translator developer. In other words, the technique maps the whole set of domain-specific constructs captured in the metamodel to the given programming language. This results in a ”debuggable” model, as the mapped constructs become available in an Integrated Development Environment (IDE) including the debugger. It also enables automatic domain-specific code completion further decreasing development time.

II. Background

During the design process of a domain-specific language in a metamodeling environment, a set of provided modeling constructs can be used. Each modeling tool has a different set of constructs that serve as the building blocks of the metamodeling language and in-turn all domain-specific languages defined with the tool. These building blocks are usually hard-coded into the systems. Most tools are capable of defining the metamodeling language using *itself*. This metamodel is referred to as the meta-metamodel. The meta-metamodel of GME is illustrated in Figure 1.

Model items are defined as First Class Objects (*FCOs*), but can be differentiated by the actual type of the class. (*FCO* is marked as abstract.) A simple model item is represented by an *Atom* node. If structurally hierarchical models need to be created, *Models* are used that can contain other elements. Relations among models can be expressed by *Connections* linking two objects together that have the

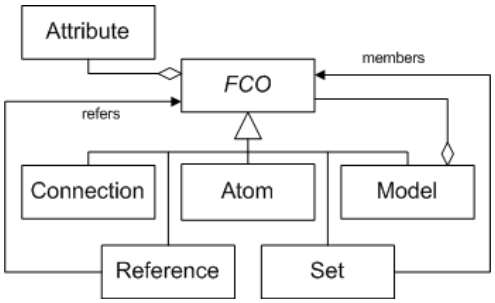


Figure 1. Part of the meta-metamodel of GME

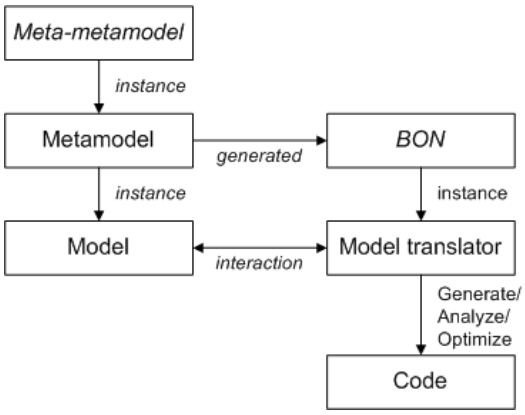


Figure 2. Schematic overview

same parent model, *References* linking two objects together that may have separate parents and *Sets* that link several objects together within the same parent again. Properties of model items are represented by *Attributes*. Apart from these constructs, a few others exist for constraint handling and visualization of the models.

Model translators that perform analysis or interpretation of domain-specific models can be considered extensions of a modeling environment. The modules cannot be built into the environment as the domain is not available at the time of the creation of the generic tool environment. In other words, the modeling language and the corresponding translator are created by the user of the generic metaprogrammable tool and not its developer. However, modeling environments need to facilitate the creation of these extensions. In GME, this is solved by the utilization of COM components [4]. The model translator needs to implement a given COM interface (*GmeLib.IMgaComponentEx*) and then it can be registered into the system. These software modules can access the model elements through other COM interfaces. With this technology the model items can be reached via a generic, domain-independent and low-level API. That is, non-typed wrapper classes are used. Another, more natural way of handling models, is via a high-level, domain-specific interface. This allows the programmer to see typed wrapper classes with names that match the names in the metamodel. This requires a generation process that creates the appropriate class structure based on the metamodel. GME supports non-typed model interpreters in C++, Java, Basic and Python; furthermore, typed interpreters can be created in C++. The class hierarchy allowing domain-specific model access in C++ is called the GME Builder Object Network (BON) Extension. In Figure 2, the schematic architecture of GME is depicted.

On the highest level, the meta-metamodel that defines the rules applying to metamodels is hard-coded into the system. The metamodel of the selected domain has to be created by the user, but the system automatically enforces the rules defined in the meta-metamodel. The domain-specific modeling environment is automatically generated from the metamodels. The corresponding model translator is hand-coded by the user, but a domain specific API utilizing a class hierarchy that helps the development is automatically generated by the system from the metamodels. The interaction between the model translator and the model is also supported by the system; the instantiation of the typed wrapper classes are done automatically on-demand. Finally, the output of the model translation can be an optimized or analyzed model, but more frequently it is program code based on the model.

III. Contribution

When creating a model translator, the modeling environment is extended with a layer that makes model elements reachable from a given platform or language. Our goal is that the models developed in GME be available from any Microsoft .NET platform language. Specifically, .NET C# [5] has been chosen as target language, however, the approach is general enough not to restrict the solution to this programming language.

A. Domain-Independent Model Translator API

Naturally, developing a non-typed model translator API is much easier, because in this case only wrapper classes that utilize the general API are used. This general API use *String* method parameters, corresponding to the names of domain-specific concepts, to handle the different domains. GME utilizes COM technology [4] that is widely supported by any .NET language. Thus, a non-typed (i.e. domain-independent) model translator API can be automatically generated from the IDL descriptors of GME interfaces. Also, the skeleton of the translator can be provided. Figure 3 depicts the main code segments of such a C# translator. The translator has to implement several properties that are used by the environment (such as `ComponentProgID`) and an entry point for the user code (`InvokeEx`).

B. Domain-Specific Model Translator API

For automatically generating a domain-specific model translator API, one has to handle the different constructs

```
[ClassInterface(ClassInterfaceType.AutoDual)]
public class Interpreter : GmeLib.IMgaComponentEx,
       GmeLib.IMgaVersionInfo {
...
  public string ComponentProgID {
    get {
      RegistrationServices rs = new
                          RegistrationServices();
      return rs.GetProgIdForType(GetType());
    }
  }

  public void InvokeEx(GmeLib.IMgaProject proj,
     GmeLib.IMgaFCO curren, GmeLib.IMgaFCOs selected,
     int param) {
    GmeLib.IMgaTerritory terr = null;
    proj.CreateTerritory(null, out terr, null);
    proj.BeginTransaction(terr,
GmeLib.transactiontype_enum.TRANSACTION_GENERAL);

    MGALib.IMgaFolder root = proj.RootFolder as
                                    MGALib.IMgaFolder;
    foreach (MGALib.IMgaFCO fco in root.ChildFCOs)
    {
      ...
    }
  }
...
}
```

Figure 3. Non-typed interpreter code segment

supported by the modeling environment and the language. GME allows domain specific language constructs similar to the language elements of C++. These partly differ from the ones used in C#, namely C# does not allow the use of multiple/interface/implementation inheritance. Also, in GME, there are some modeling constructs that do not exist as language elements, such as association classes, sets or references. Therefore, a projection is needed to map the modeling constructs to some programming language elements.

1) Class Hierarchy: Obviously, concepts in the modeling language should be mapped to similar construct in the programming languages if one exists. For example, classes should be created for metamodel nodes. This statement seems to be logical, but a careful examination of the meta-metamodel reveals that it is not possible. The problem lies in the different inheritance types that are supported by GME. Figure 4 depicts the inheritance part of the MetaGME model.

In GME, a sophisticated inheritance operator set [6]

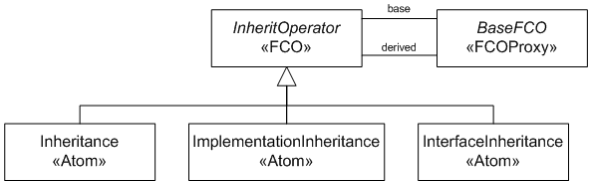


Figure 4. Inheritance definition in GME

(Figure 4) has been provided. Next to the general inheritance, two additional operators are available to provide finer-grained control over the inheritance relation. Implementation inheritance propagates all of the parent's attributes, but only the containment association, where the parent functions as the container, to the child type. No other associations are inherited in this case. Interface inheritance allows no attribute inheritance, but does allow full association inheritance with the exception of containment relations where the parent functions as the container are not inherited. Note that the union of the two special inheritance operators gives the common inheritance, and their intersection is null.

If only single inheritances had to be supported, normal class inheritance would be enough in the generated code. However, GME facilitates multiple inheritance that makes the generation of the C# equivalent code more complex. As C# only supports single class inheritance and multiple interface implementations, at most one model element can be translated into a class, others involved in the multiple inheritance need to be transformed into interface-class pairs. Naturally, it is easier to handle the model elements uniformly; therefore, each model element should be transformed into an interface-class pair. Naming conventions in the generated code is a minor decision, but the user of the system will have to be familiar with it. Also, the generated code needs to be natural for the developer, thus, naming has a huge impact on system usability. We have found that naming the interface and not the implementation class as the model element creates a more natural development environment.

As mentioned, each model item requires an interface-class pair, or more precisely, the model elements that are parents to other items need the pair, items that are only derived parts in an inheritance relationship can be translated into classes. Unfortunately, the described solution generates implementation classes with the same code fragments. This code repetition could lead to programming faults in a software development process. However, in this case the program code is automatically generated; therefore, we can assume that in the end, it works as expected and does not require any modification from the end-user. Figure 5 summarizes the possibilities.

Another solution for allowing multiple inheritance in C# has been studied, but that would lead to a less natural output code that would not resemble the meta. The solution given in [7] requires the same number of generated classes, but the parent-child relationship is not solved with inheritance but a combination of member variables, inheritance and implicit conversion operators. This would naturally be an appropriate solution in our case, but the complexity of both the generator and the output code would increase. Furthermore, none of the given disadvantages of the first solution applies to our case.

To support the three types of inheritance operators, additional interface indirections have to be introduced in the

generated C# output. Namely, a new interface is required for each type of operator, which can be implemented by the derived classes. Unfortunately, the three inheritance operators can be used orthogonally, thus, there are cases when a model item is translated into three interfaces and a class as illustrated in Figure 6.

2) Connections: In our solution, each association class is translated into a C# class, that contains the member variables. With the name of the connection, the class can be queried from the classes that are in the relationship. Also, as navigation between model items should be based on the role names of the connections, a navigation method is also generated into the output. The navigations generated into the C# code are based on the `IEnumerable` .NET interface, which does not allow insertion or addition to these collections. To create a new connection, the user needs to create a new instance of the connection class and then use its `Connect` method with the appropriate parameters. Figure 7 illustrates the generated code for a simple association.

As we study the meta-metamodel of GME, we note that association classes are derived from FCOs, which were included in the inheritance relationships. Thus, there can be inheritance relationships between association classes. (Having inheritances between association classes and other types are not supported, and probably meaningless.) Naturally, derived association classes inherit all the attributes, constructors and `Connect` methods from their parent and extend these with their own members. In Figure 8, a complex inheritance example is depicted with the generated C# code.

3) Attributes: In GME, three types of attributes can be created: Boolean, field and enumeration. In the generated classes, all of them are translated into C# properties that can query or set the values. In addition, the enumeration definitions are created in a separate class to allow reusing them.

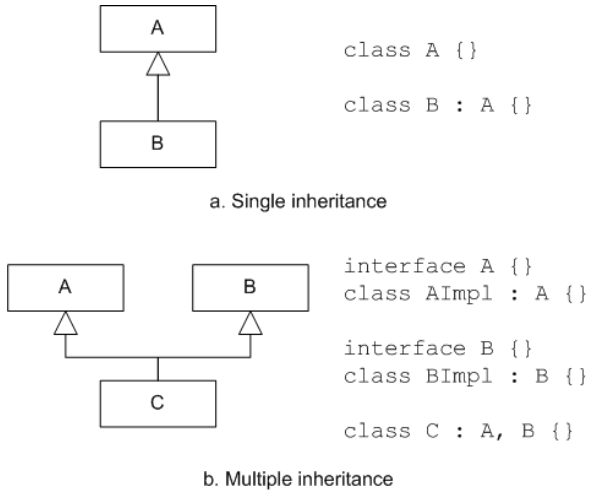


Figure 5. Inheritance and generated code

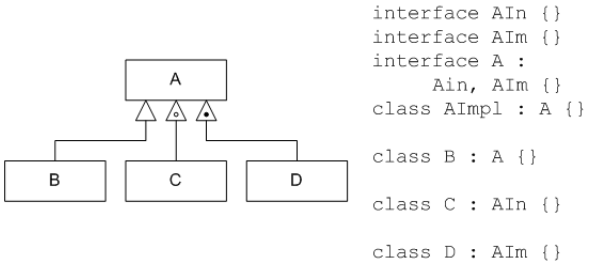


Figure 6. Inheritance and generated code

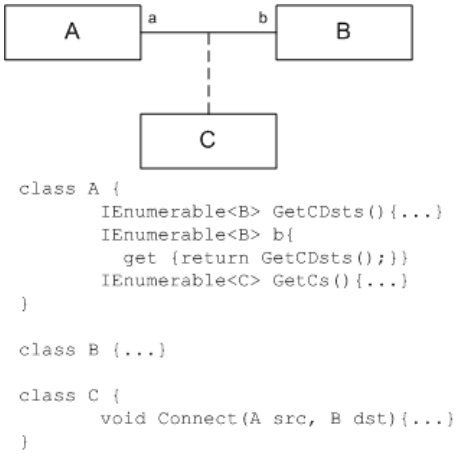
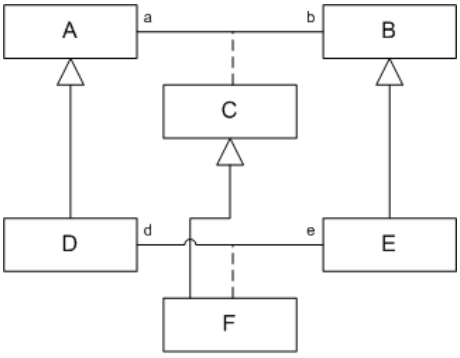


Figure 7. Simple connection

4) Sets, References, and Models: Modeling concepts that are unique in a modeling environment can only be translated into individually constructed code. A set is a collection of objects, while a reference is a pointer to another object. In GME, models are special model elements that can contain other items, thus, this enabling hierarchical models. These concepts differ in their meaning, however, after studying the corresponding parts of the meta-metamodel, we can find that their hierarchical representations are identical (see Figure 9). This means that the processing of these functionalities should be handled similarly. Differences are only given by the member names and types that have to be generated.

However, adding elements to a model differs from adding an element to a set, because GME handles model containment as a more integral part of modeling. Containment can be viewed as an essential concept in GME, every object (except the root folder) has a container. By default, GME provides a `RootFolder` for each of its models or metamodels, this is the topmost containment level in a model. Apart from this, all the model elements have their containers, thus, in the generated code, we have made a container parameter required in the constructors of model element class.




```
class AImpl : A {...}
...
class D : A {
       IEnumerable<B> GetCDsts(){...}
       IEnumerable<C> GetCs(){...}
       IEnumerable<E> GetFDsts(){...}
       IEnumerable<F> GetFs(){...}
       ...
}

class F : C {
       void Connect(A src, B dst){...}
       void Connect(D src, E dst){...}
}
```

Figure 8. Association class inheritance

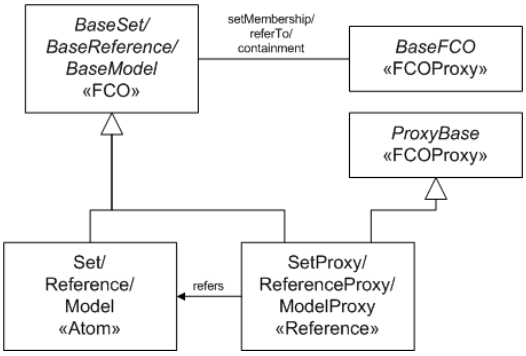


Figure 9. Sets, References and Containments in MetaGME

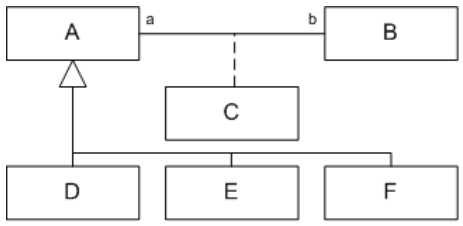
5) Class Hierarchy Revisited: Now that the main modeling concepts have been mapped to programming constructs, we have to consider the class hierarchy from another point. We have already introduced how the items are transformed into class hierarchy, but how these objects could be referenced needs to be studied. When a class does not have a child item, we do not face any problem. However, with child items, the polymorphism [8] of the modeling environment has to be handled.

In the generated code, a problem arises when classes corresponding to parent nodes need to be instantiated. Namely, when a parent item is linked to the currently processed element, the parent interface has to be used as the return type in the generated C# output. But in the body of the method, we need to differentiate between the derived elements to be able to return the child class instance of the given element. This means that we have to handle items with and without child nodes differently not only when the classes corresponding to them are generated, but also when those classes are instantiated. With the given method the polymorphism of the modeling environment is carried out in the code as well. Figure 10 illustrates the described solution.

When the association defined to class B is processed, the return type of the generated method would use the class corresponding to the item on the other side of the connection if there were no child nodes included in the figure. However, as there are derived nodes in this case, the interface have to be used to allow polymorphism in the code. On the other hand, in the method body, the actual classes have to be instantiated, thus, the code needs to check the actual type of the queried element and return an instance of the appropriate class. The generated code utilizes lazy loading of the model elements, thus cloning any cached object is not an option in this case. (The illustrative code in Figure 10 uses the *yield* C# keyword, which provides a value to the enumerator return object.)

6) Processing Algorithm: The processing algorithm that generates typed model interpreter can be divided into two parts. Firstly, the elements found in the metamodel has to be traversed and processed. Secondly, from the parsed information, the C# output has to be generated.

In the first phase, all the model items are traversed from the `ParadigmSheets`. During the traversal, wrapper classes that represent the objects are generated. In the second phase, each of the wrapper classes is processed and the code generation is executed separately. Attribute, association, containment and inheritance information of the model elements is queried on-the-fly. Thus, when a class




```
class AImpl : A {...}
class D : A {...}
...
class B {
    IEnumerable<A> GetCSrcs() {
       switch(INSTANCE_MODEL_TYPE) {
        ...yield return new AImpl(...);
        ...yield return new D(...);
        ...yield return new E(...);
        ...yield return new F(...);
       }
    }
    ...
}
```

Figure 10. Polymorphism support

representing a model item is generated into the output, the surroundings of its wrapper class (in the wrapper object network) is also used.

Algorithm 1 shows the pseudo code of the algorithm.

Algorithm 1 Processing algorithm

```
1:  procedure Generate(root : RootFolder)
2:  for all model in root.getAllParadigmSheets() do
3:      ProcessParadigmSheet(model)
4:  for all wrapper in General.Wrappers do
5:      wrapper.GenerateCode()
6:
7:  procedure ProcessParadigmSheet(model : Model)
8:  for all object in model.ChildObjects do
9:      if object.Type = ATOM then
10:         CreateAtomWrapper(object)
11:     else
12:         ... {handle other model element types}
```

7) C# BON in Action: This section illustrates a simple hierarchical state machine language and the generated API for model translator development. The corresponding metamodel depicted in Figure 11. Note that our state machine metamodel contains a *State* that can contain substates and a *Transition* association that provides links between the *States*. Start states in the FSM model are differentiated by a Boolean attribute. The main segments of the generated classes, such as the members, the body of an attribute retrieval and the body of a connection query, are shown in Figure 12.

In Figure 13, the main method of the model processor is depicted. In this example, we check whether each of our states contains exactly one *StartState*. The code depicts that even the root folder (the top level container) has its equivalent generated class. This is the primary connection point between the model and the code.

Finally, in Figure 14, the recursive checking functions are depicted. Notice that the generated domain-specific API makes it trivial to write translators since it hides all the tool-specific details on how to access the generic data structures representing the models.

IV. Conclusion

Metaprogrammable modeling tools, such as GME, make it easy to develop domain-specific modeling languages. This simplicity should be transferred to model translator creation as well. In this paper, a general method has been illustrated to allow the generation of a domain-specific API from the metamodels that describe the corresponding domain-specific modeling language. The given process not only maps the

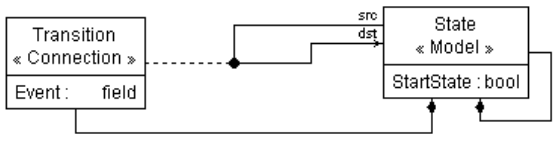


Figure 11. Finite State Machine metamodel

```
public class Transition : IConnection
{
  internal IMgaConnection mgaObject;
  internal static IMgaMetaConnection mgaMeta = ...;

  internal Transition(IMgaConnection mgaObject)
    {/*set members*/}
  public static Transition CreateNew(State parent)
    {/*create MGA object, return new Transition*/}
  public void Connect(State obj1, State obj2)
    {/*connect two States*/}

  public string Event{ get{...} set{...} }

  public State Src { get{...} set{...} }
  public State Dst { get{...} set{...} }
  public State Container{ get{...} }
}

public class State : IModel
{
  internal IMgaModel mgaObject;
  internal static IMgaMetaModel mgaMeta = ...;

  internal State(IMgaModel mgaObject)
    {/*set members*/}
  public static State CreateNew(RootFolder parent)
    {/*create MGA object, return new State*/}
  public static State CreateNew(State parent)
    {/*create MGA object, return new State*/}

  public bool StartState{ get{...} set{...} }

  public IEnumerable<Transition> GetTransitions(){...}
  public IEnumerable<State> GetTransitionDsts(){...}
  public IEnumerable<State> dst{ get{...} }
  public IEnumerable<State> GetTransitionSrcs(){...}
  public IEnumerable<State> src{ get{...} }
  public IEnumerable<State> ContainedStates{ get{...} }
  public IEnumerable<Transition> ContainedTransitions{
    get{...}
  }
  public IContainer Container{get{...}}//State or RootFolder
}
```

Figure 12. Generated classes to the FSM metamodel

```
MGALib.IMgaFolder root = project.RootFolder as MGALib.IMgaFolder;
RootFolder rf = new RootFolder(root);
foreach (State state in rf.ContainedStates)
{
    if (!RecursiveCheck(state))
    {
        //handle error
        state.
    }
}
```


Figure 13. Main method of the interpreter

```
private bool HasExactlyOneStartState(State state)
{
    return state.ContainedStates.Where(s => s.StartState).Count() == 1;
}

private bool RecursiveCheck(State state)
{
    if (!HasExactlyOneStartState(state))
        return false;

    foreach (State substate in state.ContainedStates)
    {
        if (!RecursiveCheck(substate))
            return false;
    }

    return true;
}
```

Figure 14. Verifier methods in the interpreter

domain-specific constructs to a programming language, but it also generates a program representation of the designed models. Thus, the modeling concepts and wrapper objects of the models themselves become available during programming. The generated code allows the interaction between the modeling environment and the application code, as it can be used to traverse the model hierarchy and also to modify or create model items. This extension to the modeling environment gives a more natural development environment to the users as all their usual development tools can be utilized to analyze the code representation of the model.

ACKNOWLEDGMENT

Tamás Vajk would like to thank the Institute for Software Integrated Systems at Vanderbilt University for providing a summer internship to him.

REFERENCES

[1] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional, Second ed., August 1999.

[2] S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society, March 2008.

[3] "Generic Modeling Environment Website." http://www.isis.vanderbilt.edu/projects/gme/.

[4] A. Nathan, *.NET and COM: The Complete Interoperability Guide*. Sams, February 2002.

[5] D. S. Platt, *Introducing Microsoft .NET*. Microsoft Press, Third ed., May 2003.

[6] Á. Lédeczi, M. Maróti, and P. Völgyesi, "The Generic Modeling Environment," tech. rep., Institute for Software Integrated Systems, Vanderbilt University, May 2007.

[7] D. Esparza-Guerrero, "Simulated Multiple Inheritance Pattern for C#." Code Project, 2005. Design pattern for simulating multiple inheritance in C#.

[8] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, November 1994.