ANEMIC: Automatic Interface Enabler for Model Integrated Computing

Steve Nordstrom, Shweta Shetty, Kumar Gaurav Chhokra, Jonathan Sprinkle, Brandon Eames, and Akos Ledeczi

Institute for Software Integrated Systems, Vanderbilt University
2015 Terrace Place, Nashville, TN 37235
{steve.nordstrom,shweta.shetty,kg.chhokra,
jonathan.sprinkle,b.eames,akos.ledeczi}@vanderbilt.edu
http://www.isis.vanderbilt.edu

Abstract. A domain-specific language provides domain experts with a familiar abstraction for creating computer programs. As more and more domains embrace computers, programmers are tapping into this power by creating their own languages fitting the particular needs of the domain. Graphical domain-specific modeling languages are even more appealing for non-programmers, since the modeling language constructs are automatically transformed into applications through a special compiler called a translator. The Generic Modeling Environment (GME) at Vanderbilt University is a meta-programmable model-ing environment. Translators written to interface with GME models typically use a domain-independent API. This paper presents a tool called ANEMIC that generates a domain-specific API for GME translators using the same metamodel that generates the language.

1 Introduction

One of the disadvantages of low-level coding is that a small change in the requirements of the program could necessitate drastic changes in the code. Consider the Y2K challenge of the late 1990s: the size of the requirement change - change the year representation from two digits to four - was small, yet a vast amount of effort was needed to correct this change. One solution that helps to alleviate the disparity between requirements change and implementation change is to generate the code of the final system from a centralized set of models. This technique, called Model-Integrated Program Synthesis (MIPS), is an application of Model-Integrated Computing (MIC) [1].

In order to generate final system code, several transformations take place. Typically, the set of models are examined and interrogated using a specialized high-level compiler (referred to in this paper as a *translator*) and the output of that translator is the final system code. However, the process of creating this translator is currently heavily dependent on the programmer and little advantage is taken of code generation. A great deal of the work that the programmer does can be streamlined and automated using user-defined macros and traditional

F. Pfenning and Y. Smaragdakis (Eds.): GPCE 2003, LNCS 2830, pp. 138–150, 2003.
© Springer-Verlag Berlin Heidelberg 2003

"code-cloning", but this process is error prone, and inconsistent across users. Furthermore, the macros operate on the individual class level, and do not have a global view of the software architecture.

This paper presents a tool that automatically generates an appropriate domain-specific API from the metamodels that describe the domain specific modeling language.

2 Background

MIC is a framework for developing domain artifacts for computer-based systems. MIC depends on a model-based definition of the target system, and it integrates the created models when generating domain-specific artifacts. The same models are often used by external analysis and simulation tools to verify properties of the system under development. Example uses for MIC are the generation of real-time schedules from software models, creation of a configuration file to integrate distributed systems, or the generation of source code that is later compiled to execute within an embedded system.

MIC relies heavily on the use of domain-specific languages to describe the final system implementation. A domain-specific language allows a modeler to describe the system in terms of the domain rather than in terms of traditional computer languages. Domain-specific modeling environments (DSME's) are the interface for domain experts to program using a domain-specific modeling language (DSML). The DSME provides domain-specific constructs that are associated with one another to describe a computer-based system. For further explanation of this process please refer to [2].

However, modeling language development is not a trivial task. In addition to the development of the language ontology, syntax, well-formedness and semantics, there is also the question of representation and implementation.

2.1 Metamodeling

A technique called *metamodeling* can be used to rapidly describe the syntax and static semantics (well-formedness) of a language. The artifact of the metamodeling process - called the metamodel - is generally retained in an object database for further manipulation, and can later be modified to generate a new version of the domain-specific language.

The metamodel is a formalized description of a particular modeling language and is used to configure the Generic Modeling Environment (GME) (GME is a configurable toolset that supports the easy creation of domain-specific modeling and program-synthesis environments [3]). Here, we provide a brief overview of the metamodeling concepts. For a more in depth discussion, see [4].

Using essentially the UML class diagram and OCL syntax, metamodels describe the entities, their attributes, and inter-relationships that are available in the target modeling environment [5] [6] [7]. The DSML may be specified in terms of Models, Atoms, Connections, References, and Sets. Models are the centerpieces of the MIC environment. They are hierarchically decomposable objects

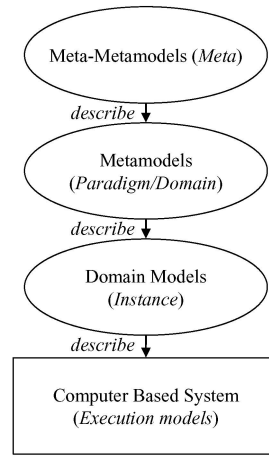


Fig. 1. The four layers of metamodeling

that may contain parts and inner structure. Atoms cannot be further decomposed. Associations between objects are captured by Connections, References, and Sets. References can be used to associate objects in different parts of the model hierarchy. All objects may be qualified by one or more Attributes: key-value pairs that capture non-structured information.

The metamodel in Figure 2 captures a language used to model the composition of neighborhoods. It shows that each Neighborhood may contain zero or more Buildings, where each Building may either be a House or a Store. Each House may contain Residents, and each Store may contain Patrons. The Neighborhood may also contain Walkways that connect the Buildings to each other.

Once a metamodel has been created and the domain-specific language generated, only a portion of the overall domain-specific environment is complete. Without a compiler, the domain-specific language is just pictures - the compiler gives semantics to the syntax specified by the metamodel. With respect to Figure 2, if we want to find all the instances of Walkways that connect one Building to another we would need to write a special kind of translation program to extract this information from the model.

In order to provide the semantics, syntax patterns are transformed into a domain artifact (e.g., glue code, configuration files) with a special compiler called a *translator* [1]. The translator provides the mapping from the domain-specification in the language to the domain-use in the application.

2.2 Language Translation

It is the translator that releases the full power of a domain-specific language. The translator abstracts away the mundane details of the implementation by encoding domain details that can be automatically generated rather than burdening the modeler with the task of manually implementing them. In this way,

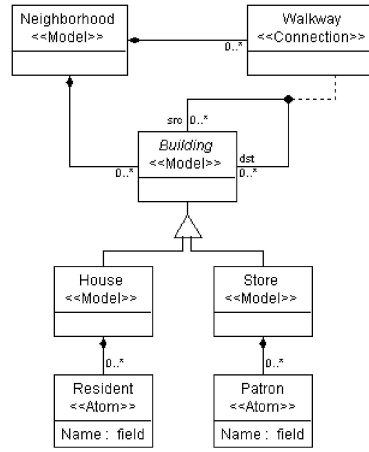


Fig. 2. A UML description of a domain-specific modeling language, used to model Neighborhoods. A Neighborhood may be modeled as one or more Buildings, connected by Walkways. The Houses or Stores (both being Buildings) may contain Residents or Patrons, respectively

models can be interpreted (translated) in more than one way, while the meaning of the models is captured only once. A good example of this is the generation of either C++ or Java classes from the same UML class diagram [8] or executable code from Statecharts [9].

The creation of the translator is strongly connected to the domain-specific modeling language that the translator compiles. Since the translator is the connection between the domain-specific implementation of the models and the (usually) domain-independent implementation of the computer based system (i.e., many computer-based systems are implemented in a programming language, such as C++ or Java, which require the management of details not important to the high-level design of the system) a great deal of domain knowledge must be present in the translator. For the GME, a translator is created in a standard language (VB, C++, C#, etc.) and operates on the models through a public interface that utilizes COM.

The translator takes as input a model database, and produces as output the appropriate domain-specific artifact. The translator can make execution decisions for generation through the types of models that it encounters by querying the runtime-types of the objects. These types are the same as those described by the metamodel of this domain-specific language. The next two sections describe the existing framework for interfacing with domain-independent model types, and the new framework for automatically generating specialized classes that extend these domain-independent model types to create domain-specific classes.

3 The Builder Object Network (BON)

As mentioned in the preceding section, the process of translation involves querying the model database for the types of entities defined and their relation (hierarchy, aggregation, association etc.). Figure 3 shows a schematic of GME's modular COM-based architecture; the GME core components (GModel and GMeta) expose a set of COM interfaces that facilitate model interrogation (For a more in-depth discussion of the architecture, refer to [2]).

While the COM interface provides the translator writer with all the functionality needed to access and manipulate the models, it entails using repetitious COM-specific querying, error checking and handling. To abstract these issues from the translator writer, GME provides a collection of C++ wrapper classes: the Builder Object classes.

The fundamental types of entities in GME (Atoms, Models, Sets, Connections and References) share common traits. Each modeling entity has an associated name, kind name, metamodel and type. They may, depending on the metamodel and the specific model being examined, each be qualified by one or more attributes, connected to one or more objects. The operations of querying and specifying such details are common to these entities and are thus abstracted in to a base class called `CBuilderObject`.

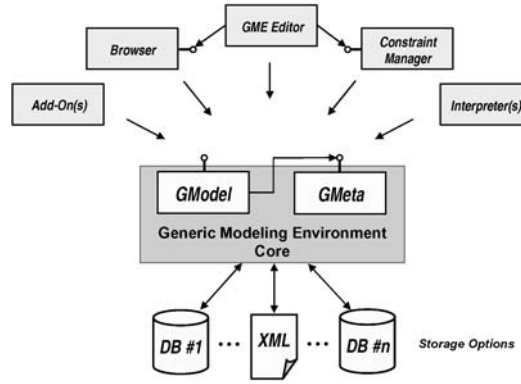


Fig. 3. GME Architecture provides a generic, meta-programmable modeling environment upon which domain-specific models can be created and examined

Many operations are consistent across all types of entities in the BON, which justifies the `CBuilderObject` base type. This important class serves as the base type for any BON class and provides a default implementation for the most common tasks. The corresponding classes for Atoms (`CBuilderAtom`), Models (`CBuilderModel`), Sets (`CBuilderSet`), Connections (`CBuilderConnection`) and References (`CBuilderReference`) specialize `CBuilderObject` via inheritance to provide relevant functionality. For example, `CBuilderModel` adds to

`CBuilderObject` the ability to query or create contained entities. Figure 4 illustrates the inheritance relationship between the various BON classes.

When the user initiates model translation, the component interface builds a graph mirroring the models: for each model, atom, reference, set and connection an object of the corresponding class is instantiated. We refer to this graph as the Builder Object Network and the infrastructure for creating it as the Builder Object Network API (BON). While the BON implements a corpus of access and manipulation methods, it must by design be generic: it has no knowledge of the attribute names, kind names or other qualifiers of the entities defined by the meta-model. Thus to access a particular attribute, say "Color", of an Atom, the translator writer provides the domain-specific intelligence via parameters to the appropriate method.

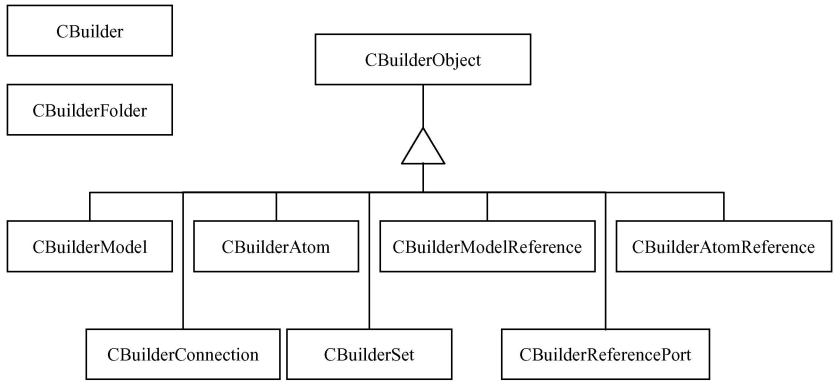


Fig. 4. Builder Object class hierarchy dictates that all Models, Atoms, Connections, and References inherit from the `CBuilderObject` class

To allow for domain specific information, the BON provides a mechanism for extending the general-purpose functionality of the `CBuilder` classes. For more functional components, the `CBuilder` classes can be extended by the programmer with inheritance [2]. By using a pair of supplied macros, the programmer can have the component interface instantiate these paradigm-specific classes instead of the default ones. This allows the programmer to have the implementation classes more closely mimic the properties of the metamodel.

Consider the metamodel presented in Figure 2. With the default BON methods, the following code would be needed to retrieve and iterate through a list of Residents given a House.

```
extern CBuilderModel *pHouse;
const CBuilderAtomList*pAtomResidents =
   pHouse->GetAtoms("Resident");
if(pAtomResidents && !pAtomResidents->IsEmpty()){
```

```
   POSITION pos = pAtomResdients->GetHeadPosition(pos);
   while(pos){
      CString haircolor;
      CBuilderAtom* pResident =
         pAtomResidents->GetNext(pos);
      VERIFY(*pResident.GetAttribute(
             "haircolor", haircolor));
   }
}
```

The programmer could specialize both House and Resident, such that the extension classes meet this requirement. Let `CSBuilderHouse` and `CSBuilderResident` extend `CBuilderModel` and `CBuilderAtom` via inheritance respectively. Thus the above code is transformed as follows:

```
extern CSBuilderHouse *pHouse;
const CSBuilderResidentList*pResidents =
   pHouse->GetResidents();
if(!pResidents && !pResidents->IsEmpty()){
   POSITION pos = pResidents->GetHeadPosition();
   while(pos){
      CSBuilderResident *pResident =
         pResidents->GetNext(pos);
      CString haircolor =
      pResident->GetAttribute_haircolor();
   }
}
```

From the above code, it is apparent that the extension leads to much smaller and more intuitive code. The extension eliminates the need for the programmer to concentrate on error checking and type manipulation code.

Moreover, the default access methods being generic in BON, do not provide specific type checking. For example, for the `GetInConnections(Cstring& name, CBuilerObjectList &list)` method, if the wrong name is provided or if the string name provided is case-incongruent with the name defined by the metamodel, the function returns an unexpected false. The translator writer must pay close attention to the correctness of the name and return value by either memorizing the metamodel, or constantly referring back to it. Such discrepancies manifest themselves as run-time errors (`VERIFY` failures), which are difficult to detect and debug. Furthermore, once a list of connections is retrieved, the programmer must manually implement repetitious code to distinguish the various connections found. By extending the default `CBuilder` classes, this onus of checking details may be moved from the programmer to the compiler by implementing a function `GetWalkwayInConnections()`that returns a valid list of only `CSBuilderWalkway` objects.

This ability to augment the default implementation with the desired functionality can be exploited by generating paradigm-specific extensions of the

BON. The translator writer typically generates customized BON classes using the metamodel as a source for architecture, naming convention, and as a reference for parameters to the generic BON. The automatic generation of such a class hierarchy directly from the metamodel would, therefore, facilitate creation of the language translator by decreasing the time spent in class construction, ensuring correctness of the extensions, and allowing the programmer to devote his creative energies towards the specification of semantics.

4 Domain Specific API Generation

The ANEMIC (Automatic Interface Enabler for Model Integrated Computing) tool was created to perform the automatic generation of the domain specific API used to create a language translator. ANEMIC generates C++ classes to implement this specialized BON class structure directly from the metamodel. The ANEMIC tool is itself a model translator that traverses the metamodel to produce a code framework for the domain translator, consisting of classes and methods corresponding to the entities captured in the metamodel, as well as the methods for traversing the domain models.

4.1 ANEMIC Translator Approach

The ANEMIC translator is a transformation program that traverses the network of objects in the metamodel. Figure 5 shows the decomposition of the ANEMIC translator. In [10] and [11], algorithms for writing structured model translators are discussed in detail.

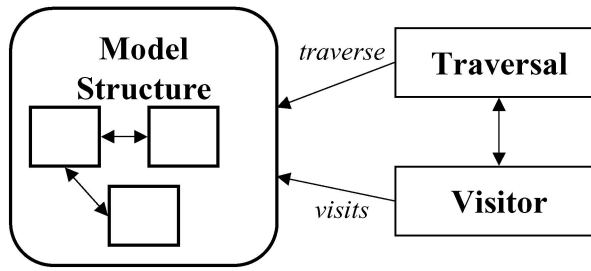


Fig. 5. Design of the ANEMIC translator includes automated visitor and traversal patterns for navigating the BON and collecting specialization information

- *Model Structure:* The model structure specification defines what classes of objects are available, and their inter-relations (compositions, hierarchies, associations, etc.). This information is captured in the metamodel being examined.
- *Traversal:* Traversal captures how the models should be traversed. The specification addresses the order of traversal, which can vary based on the type of

entities encountered. A breadth-first search strategy is used here, which is a graph search algorithm that tries all one step extensions of current paths before trying the larger extensions. The root is examined first, followed by the children of the root, and then the children of those nodes are examined, etc. This helps in keeping the class dependencies and also preserves the hierarchy information.

– *Visitor:* Visitors capture the actions to be taken when visiting an object of particular type. The different types of objects would be "Model", "Atom", "Reference", "Connection", "Attribute", "Inheritance" etc. The visitor considers - based on the type of the current object - what state information should be stored to produce an accurate architecture. ANEMIC gathers the structural (containment, hierarchy, connection, inheritance, etc.) and type (kind name, role name, attributes etc.) information for each entity defined.

For each entity defined, ANEMIC generates a corresponding extension class from the appropriate `CBuilder` class. ANEMIC preserves the inheritance relationships captured in the metamodel through the use of C++ inheritance structure in the generated output. If an object in the metamodel is defined abstract then, a corresponding abstract C++ class is created. This affords the programmer the ability to exploit, in code, the hierarchical relationships existing in the metamodel.

The attributes belonging to a particular entity (for example a Model or an Atom), appear as protected data members of the respective extension class. The following section elucidates this process with an example.

4.2 Class and Method Generation Using the ANEMIC Translator

Figure 6 shows the UML class diagram that represents a Resident, which has `haircolor` as an attribute. Notice that Resident is of type `Atom`. Consequently, the generated class `CSBuilderResident` extends `CBuilderAtom`. The extension is facilitated by the use of a pair of BON extension macros: `DECLARE_CUSTOMATOM` and `IMPLEMENT_CUSTOMATOM`. BON provides such extension macros for all entities defined in the GME meta-metamodel.

To enable easy identification of the generated class and eliminate naming conflicts with existing BON objects, the generated C++ class code adheres to a nomenclature where the generated classes are named as

```
CSBuilder{Name}
```

where `Name` is the name of the object in the metamodel. All default BON access methods, except those dealing with Connections, are specialized as

```
{return_value} {FUNCTION NAME}_{name}({parameters})
```

where `FUNCTION NAME` is the default BON method, `parameters` are appropriate arguments to the new method, and `return_value` is the data type of the corresponding attribute. Figure 7 shows the specialization of

`bool CBuilderAtom::GetAttribute(CString &name, CString &value)`

as

`CString GetAttribute_haircolor() const`

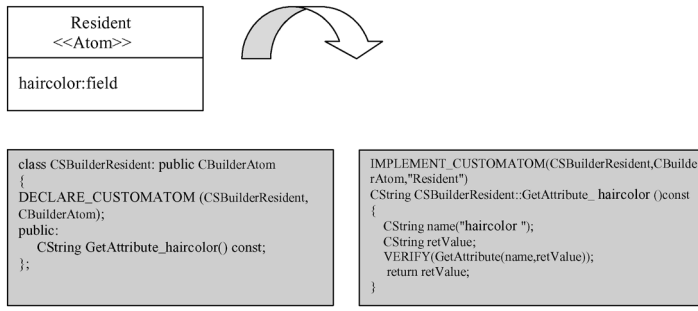


Fig. 6. ANEMIC exploits the generic `GetAttribute(name, value)` method to specifically access the attribute `haircolor`

BON methods dealing with Connections are specialized as

`{terminator_list*}{Get|Set}{In|Out}{name}Connection()`

where `terminator_list` is a typed list as specified by the metamodel and name is the name of this Connection.

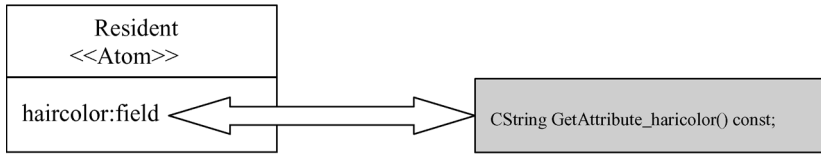


Fig. 7. Specialized methods for each attribute are generated by the ANEMIC translator using a predefined naming convention. Proper implementations for parent-class method wrappers are generated as well

The attributes of the specialized class can be of type `int`, `bool` or `CString` depending on the type specified by the user in the metamodel. Table 1 shows the mapping between GME attribute types and C++ data types.

5 Example Metamodel and Generation

Creation of a translator for the modeling language described in Figure 2 is possible with the domain independent BON API, but is now streamlined by using

Table 1. A mapping between GME attributes and generated C++ member variable types is generated by ANEMIC

Attribute Type	C++ type
<<FieldAttribute>>	`CString`
<<EnumAttribute>>	`int`
<<BoolAttribute>>	`bool`

the architecture of customized classes created by ANEMIC. By executing the API generation tool on the metamodel this required specialization takes place automatically resulting in a variety of specialized objects and methods. Two such objects are given below:

```
class CSBuilderBuilding : public CBuilderModel{
DECLARE_CUSTOMMODEL(CSBuilderBuilding, CBuilderModel)
public:
   virtual void Initialize();
   virtual ~CSBuilderBuilding();
   CSBuilderWalkwayList * GetOutWalkwayConnections();
   CSBuilderWalkwayList * GetInWalkwayConnections();
};

class CSBuilderHouse : public CSBuilderBuilding{
DECLARE_CUSTOMMODEL(CSBuilderHouse, CSBuilderBuilding) public:
  virtual void Initialize();
  virtual ~CSBuilderHouse();
  CSBuilderResidentList * GetResidents();
  CSBuilderResident * CreateNewResident();
};
```

These class definitions define the specialized BON interface to be used by the domain specific translator. Class hierarchy from the metamodel is preserved, and forward declarations are created. Also, typed lists for storing the new specialized objects and are generated. List types for containing specialized objects are also created.

```
typedef CTypedPtrList<CPtrList,
        CSBuilderWalkway *>CSBuilderWalkwayList;
typedef CTypedPtrList<CPtrList,
        CSBuilderResident *>CSBuilderResidentList;
```

Implementation of the specialized API is also generated from the metamodel by the ANEMIC tool. The following is an example method implementation from a specialized Store object, showing the generation of a proper wrapper for parent methods:

```
CSBuilderPatron * CSBuilderStore::CreateNewPatron(){
   return BUILDER_CAST(CSBuilderPatron,
           CBuilderModel::CreateNewModel("Patron"));
}
```

The automatic generation of the class definitions and method implementations now allow the domain-specific model translator to be created with much ease. For example,

```
CBuilderConnectionList GetInConnections("Walkway");
```

can be written using domain-specific methods and objects as

```
CSBuilderWalkwayList GetInWalkways();
```

By using the interface generated by ANEMIC, a translator can now be created with domain-specific objects and methods, using only a small amount of hand-written code.

6 Conclusions and Future Work

The work described in this paper significantly decreases the amount of overhead required to create an translator for a domain-specific modeling language. The domain-specific API generator can generate code that would take several days for one programmer to create, by using the metamodel that describes the language. It should be noted that the domain-specific API was typically hand created for every C++ translator before this generator was available.

In addition to the creation of the class structures, ANEMIC also takes advantage of the definition of abstract classes and inheritance hierarchies when generating the output classes. This allows translator programmers to maximize code reuse and minimize code duplication by using the same object-oriented approaches used to create the metamodels.

Future versions of this tool can be configurable by the user to follow naming conventions and personal preferences for class definition. Also, other versions of the tool that could generate an API for other languages would greatly reduce the amount of overhead for users that prefer an implementation language other than C++.

References

1. Sztipanovits J., Karsai G.: Model-Integrated Computing, IEEE Computer, vol. 30, no. 4, pp. 110-112, April (1997)
2. Ledeczi A., Maroti M., Bakay A., Nordstrom G., Garrett J., Thomason IV C., Sprinkle J., Volgyesi P.: GME 2000 Users Manual (v2.0), ISIS document, December 18, (2001)
3. Ledeczi A., Maroti M., Bakay A., Karsai G., Garrett J., Thomason IV C., Nordstrom G., Sprinkle J., Volgyesi P.: The Generic Modeling Environment, Workshop on Intelligent Signal Processing, accepted, Budapest, Hungary, May 17, (2001)

4. Sprinkle J., Karsai G., Ledeczi A., Nordstrom G.: The New Metamodeling Generation, IEEE Engineering of Computer Based Systems, Proceedings p.275, Washington, D.C., USA, April, 2001.
5. Karsai G., Nordstrom G., Ledeczi A., Sztipanovits J.: Specifying Graphical Modeling Systems Using Constraint-based Metamodels, IEEE Symposium on Computer Aided Control System Design, Conference CD-Rom, Anchorage, Alaska, September 25, (2000)
6. UML Summary, ver. 1.0.1, Rational Software Corporation, et al., Sept. (1997)
7. Object Constraint Language Specification, ver. 1.1, Rational Software Corporation, et al., Sept. (1997)
8. Jürjens J.: Formal Semantics for Interacting UML subsystems, Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002), Twente, March 20-22, (2002)
9. Harel D., Gery E.: Executable Object Modeling with Statecharts, IEEE Computer, vol. 30, no. 7, pp. 31-42, July (1997)
10. Karsai G.: Structured Specification of Model Interpreters, ECBS, pp 84-91, Nashville, TN, March, (1999)
11. Gamma E., Helm R., Johnson R.,Vlissides J.: Design Patterns: Elements of Reusable Object- Oriented Software, Addison-Wesley, (1995)