

EXCHANGE OF UML MODELS IN A DESIGN CHAIN USING A BERKLEY DATABASE

Olaf Besser, Christina Dorotska, Galina Rudolf, Bernd Steinbach
Freiberg University of Mining and Technology, Institute of Computer Science,
D-09596 Freiberg, Germany

The design chain for discrete devices includes several specialized CASE tools. There must be well defined representations of data in order to exchange models between such CASE tools. The UML is the most important modeling language for high level design. The exchange of UML models can be realized by means of an XML file that uses the XMI 2.0 standard format. We suggest in this paper the exchange of UML models using a Berkeley database in connection with a XMI support. The benefit of this approach is the faster direct access to the model elements and several simplifications in the development of design tools.

Introduction

This paper covers a special topic in the area of software representation of discrete devices. In order to generate test templates and test cases for object-oriented software we decide to use a UML model as information source of the System Under Test (SUT). The test of the object oriented software includes therefore two consecutive subtasks. First, the SUT must be modeled using a CASE tool, and second, the required test cases must be created using TEst CAse GEnerator (*GETECA*) [1]. The collaboration between these tools requires the exchange of a very complex structured large amount of data.

Several CASE tools were applied until now. The basic requirement was a well defined data format for the exchange of UML models. Initially we used the Case tool *Together* and later on *Rational Rose*. In order to import the data from the UML model to the test generation tool *GETECA* we stored the model data into a XML file using of the XMI Format.

Both CASE tools are able to store the created UML models in a file using the XMI format. However, there are differences between XML files created by these case tools. The parser developed for reading of model data for CASE tool *Together* must be adapted for the XMI files from the CASE tool *Rational Rose*. Each changing in the XML file requires the change of the parser. Besides, both CASE tools dose not satisfy the XMI standard of UML model in stored XML files.

In order to overcome these restrictions we determined to develop a new CASE tool named *UML Designer*. The tool must be conform to the UML 2.0 Standard and also provide support to the standardized XMI 2.0 format. For an iterative development of systems this tool should be able to represent the generated test templates and test cases as elements of the UML model. The output data of the *UML Designer* are taken as the input data for *GETECA* (see Fig. 1). Therefore both programs must have an interface to save or to load the data.

The using of XMI as exchange format requires implementing of two moduls for reading and for writing of the UML data. Furthermore, because *GETECA* and *UML Designer* were written in different programming languages, the reader and writer must be developed in two different programming languages. These languages are in particular Java in case of *UML Designer* and C++ in case of *GETECA*. The change of the XMI data to new standard requires the change for both read and write modules, too.

A further disadvantage of this solution consists in the access to a XMI file. The *UML Designer* saves and reads all model elements and all associated graphic information. The test tool needs only a part of the model. The XMI file is similar to a text file and must be read

sequentially. For large models these files are very extensive and the reading and writing procedures take a long period of time.

In order to save changes and adaptations of the reader and writer in the *UML Designer* and of the reader and parser of *GETECA* we decide to use a database to store the model data (see Fig. 1). An additional benefit of such a database consists in the direct access to selected model elements. Hence, the access is much faster.

In order to be open for other tools we do not exclude the application of XMI files. The data transfer from the database into a XMI file happens by a special program in the design chain named *XMI Converter*. The *XMI Converter* imports the data into the database and exports them into a XML file using the XMI format. In order to cooperate with other case Tools we follow strictly the rules of XMI format specification [2].

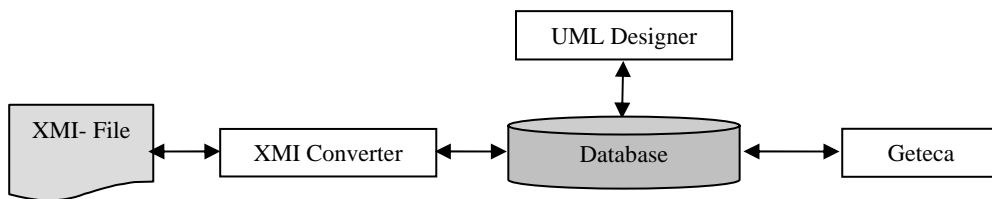


Fig. 1. Tools in design chains

The utilization of the XMI 2.0 format standard opens the doors to use our tools together with other commercial CASE tools, which support the XMI interface, too. The translation rules are described by a special file called *dbElementStructure*. This file is used from the *XMI Converter* for translation between an XMI file and the database in both directions. This file can be fitted to changed requirements at any time.

1. Fundamentals of UML, XML and XMI

The **UML** is a universal description language for all kinds of object oriented software systems. It contains thirteen different types of diagrams which are divided into two categories: six diagrams represent the static software structure, and seven diagrams visualize different dynamical aspects [3].

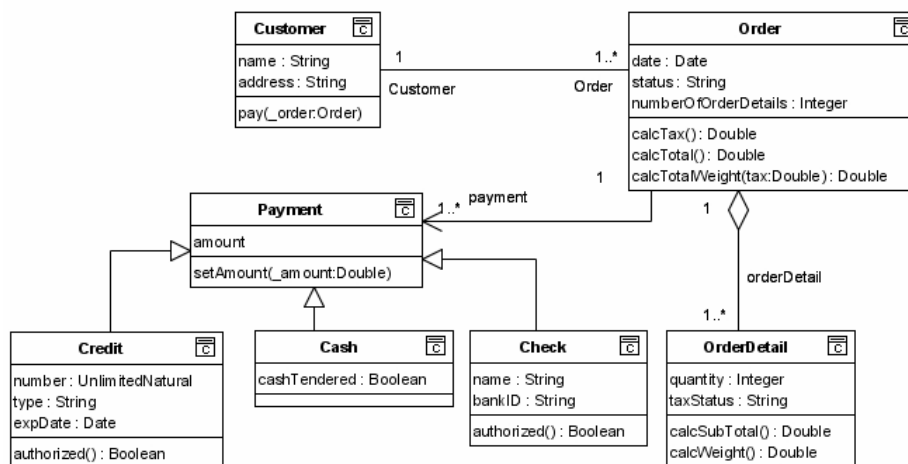


Fig. 2. UML class diagram

Basis diagram types of the UML are: use case diagrams, class diagrams, activity diagrams and sequence diagrams as well as state machines. The class diagram is the most important diagram and consists of notation elements like classes and interfaces as well as relations between them (see Fig. 2). Inside of classes their attributes and operations may be defined. The communication between several objects is described by means of activity diagrams or sequence diagrams.

In order to perform the export of the UML model into XMI data format a higher level of abstraction is required. The OMG used the hierarchical architecture of meta-models in order to define the UML. The architecture consists of four layers [3][4]. It provides a meta-meta model as the top layer, called M3 layer. This M3-model is the language to build meta-models, called M2-models. The most important example of a layer 2 model is the UML meta-model, the model that describes the UML itself.

The meta-model defines the UML. It contains three major categories of elements: classifiers, events, and behaviors. A classifier describes a set of objects. An object is an individual thing having a state and relationships to other objects. An event describes a set of possible occurrences. An occurrence is something that happens in the system and causes a consequence within the system. A behavior describes a set of possible executions. An execution performs an algorithm according to a set of rules [3] [4].

An example how the element “Class” of the UML is represented in the meta-model is shown in Fig. 3. A *Class* is a classifier that may have own features like attributes, operations, and owned classifiers. An Attribute of a *class* is represented by an instance of the class *Property*. Similar relationships exist between the class *Class* and the classes *Classifier* or *Operation*. Fig. 3 shows the mentioned part of the UML meta-model.

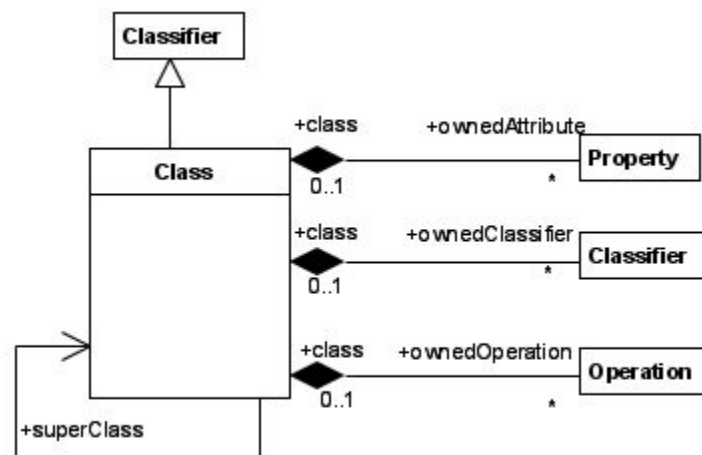


Fig. 3. Part of the UML meta-model

The **Extensible Markup Language (XML)** is a markup language that supports a wide variety of applications. XML provides a text-based means to describe and applies a tree-based structure of information. The meaning of included data is marked up by tags. An element consists of a start tag, the content, and an end tag. Thus, the whole XML file is manifested by texts, interspersed with markups that indicate both the meaning and the separation into a hierarchy of data, container-like elements, and attributes of those elements.

There are two basic approaches to map model data to XML. The first one uses textual containment of XML elements to represent their logical containment. The second one exploits the representation of containment using attributes that have an ID, IDREF or IDREFS type. We use both approaches. Child elements are put directly in the parent element in order to

indicate that parent element contains the child element. Other relationships are expressed such that the parent element gets an attribute having the same value as the identifier of the child element.

For example, the instance Customer (see Fig. 2) of the class *Class* is represented by the following part of an XMI file.

```
<ownedType xmi:type="uml:Class" isAbstract="false" isActive="false" isLeaf="false" name="Customer"
  visibility="public" xmi:id="7ecf">
  <ownedAttribute xmi:type="uml:Property" aggregation="none" isDerived="false" isDerivedUnion="false"
    isLeaf="false" isOrdered="false" isReadOnly="false" isStatic="false" isUnique="false" lowerValue="1"
    name="name" type="7f54" upperValue="1" visibility="private" xmi:id="7ebd">
    <defaultValue xmi:type="uml:LiteralString" value="" xmi:id="7eb5"/>
  </ownedAttribute>
  ...
  <ownedOperation xmi:type="uml:Operation" concurrency="sequential" isAbstract="false" isLeaf="false"
    isQuery="false" isStatic="false" name="pay" visibility="public" xmi:id="7eb6">
    <ownedParameter xmi:type="uml:Parameter" direction="in" isOrdered="false" isUnique="false"
      lowerValue="1" name="_order" type="f465" upperValue="1" visibility="public" xmi:id="7ea7">
      <defaultValue xmi:type="uml:LiteralString" value="" xmi:id="7ea3"/>
    </ownedParameter>
  </ownedOperation>
</ownedType>
```

In order to exchange XML documents, it must be sure that involved parties understand how the data are represented in XML documents. Therefore we use the XMI 2.0 standard data format.

XMI stands for XML (Extensible Markup Language) Metadata Interchange. It is the standard in order to represent object-oriented information using XML [5]. The most common use of XMI is the interchange of UML models, although it can also be used for serialization of models of other languages [6]. The main application of XMI is its use for passing models from modeling tools to software generation tools as part of model-driven engineering.

2. Storing of UML Models into a Berkeley Database

The Berkeley DB (BDB) is a high-performance, embedded database library with bindings in C, C++, Java and many other programming languages. The BDB stores arbitrary key/data pairs as byte arrays, and supports multiple data items for a single key [7]. The primary goal of the BDB is to store data ordered as a tree of structured elements. Additionally, the DBD supports the data exchange through a local network or the internet. One of the fields, where this system can be used is storage of UML models.

The Berkeley Database has a simple architecture. It does not support any schema of tables or table columns. Each dataset of BDB consists of the unique key and associated values. The size of the values is unbounded. Therefore it is possible to write the complete UML model element specification as a value. Moreover, the CASE tool *UML Designer* allocates to each element of the model an own unique identification that can be used as unique key in a dataset of the BDB. An example of the representation of the project (UML model) EXCADD07 that includes the class *Customer* and the property *name* (see Fig. 2) is shown in Table 1.

Each dataset of BDB represent one model element. As mentioned above, a model element is described by some attributes and can contain of some children. A child is a model element too and is described separately by another dataset.

Generally, the value part of the pair key/value consists of several details. The value is divided into three parts: the identification part, the specification part and the children part.

Table 1.
Representation of an UML model in the Berkley Database

	Key	Value
Project	#id-7cf8	~#id-7cf8~456e88e1:1103016e00f:-7f7c ~EXCADD07~Project~version~ ~createTime~changeTime~[Separator_Children] ~#id-7cf8#id-7f68~#id-7cf8#id-7ed0~#id-6a56~#id-7cf8#id-f465a8f:10b70c1
Class	#id-7cf8#id-7ecf	~#id-7cf8#id-7ecf~#id-7cf8#id-7f5a~ownedType~UML:Class~visibility~public ~xmi:type~uml:Class~name~Customer~isLeaf~false~isAbstract~false~isActive ~false~[Separator_Children]~#id-7cf8#id-7ebd~#id-7cf8#id-7ead ~#id-7cf8#id-7eb6~#id-7cf8#id69ddf16a-0553-4735-b516-7c765961c8c2 ~#id-7cf8#id-7375
Property	#id-7cf8#id-7ebd	~#id-7cf8#id-7ebd~#id-7cf8#id-7ecf~ownedAttribute~UML:Property ~visibility~private~xmi:type~uml:Property~name~name~isOrdered~false ~isUnique~false~isReadOnly~false~isStatic~false~isLeaf~false~isDerived~false ~isDerivedUnion~false~aggregation~none~lowerValue~1~upperValue~1~type ~#id-7cf8#id-7f54~[Separator_Children]~#id-7cf8#id-7eb5

The identification part of the value consists at first of the identification (id) of the model element. The key that is used in the BDB is the same as created by the *UML Designer*; it is repeated in the part of the value. Next, the identification of the parent model element is located. For example, the parent element of the property *name* is its class *Customer*. Further, the kind of the model element (here Project, Class or Property) and the name of the model element must be stored.

The middle part of a value consists of specifications about the model element and the last part includes identifications of the child elements. A tilde character serves as a separator.

As mentioned above, the model elements of children are stored separately as particular datasets into the database. For example the property *name* is a child element of the class *Customer*.

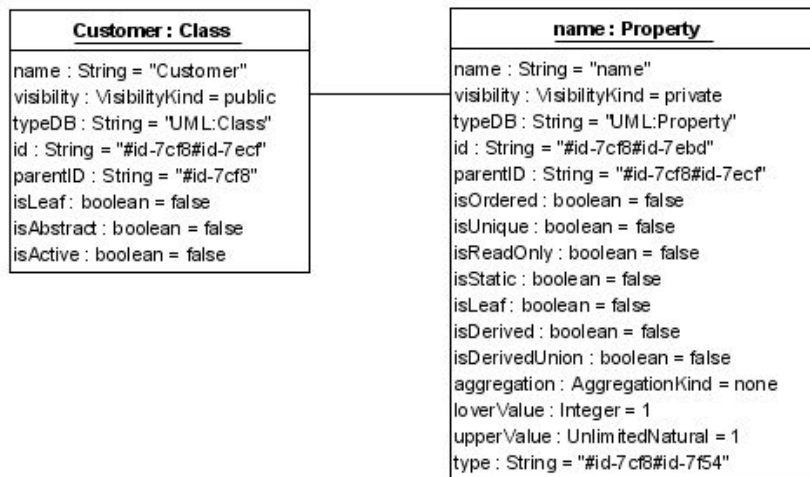


Fig. 4. Part of the object diagram representing class *Customer* and its property *name*

Using the classes from UML Meta model (see Fig. 3) we can present the class *Customer* and its property *name* by the object diagram in Fig. 4.

While reading from the database the value of a dataset is represented by a string. To be able to analyze this string more efficiently we have developed four classes represented in Fig. 5. Based on this data structure, we assure the coherence between data records and representation of the information from the value by classes of the internal UML model (see Fig. 4). These classes build an interstage between the dataset of the BDB and the UML model.

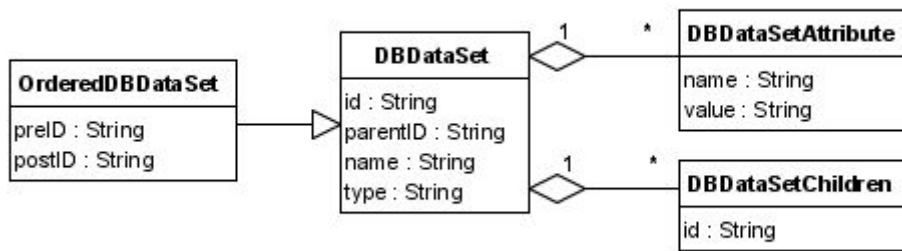


Fig. 5. Part of the class diagram representing datasets

An instance of the class *DBDataSet* represents the identification part of the value of a dataset of the BDB. Each model element has an own unique identification and an identification of the associated parent element. This allows storing the data in a tree structure and performing easier navigation through the whole UML model. Sometimes it is necessary to define an order among elements. We accomplish this requirement by means of the class *OrderedDBDataSet*. This class contains additionally identifications of the previous and of the next element in the order.

An instance of the class *DBDataSetAttribute* defines the specification of a model element and represents an attribute of the specification part of value. The property *name* of this class represents the name of an attribute (for example “visibility”) and the property *value* includes value of this attribute (for example “public”).

Instances of the class *DBDataSetChildren* contain the property *id* and describe child elements. They are associated in the children part of the value of a dataset.

We store the data of the value part of a dataset using the defined *DBDataSet* structure after the reading and parsing.

Fig. 6 shows the object diagram of the data structure for class *Customer*.

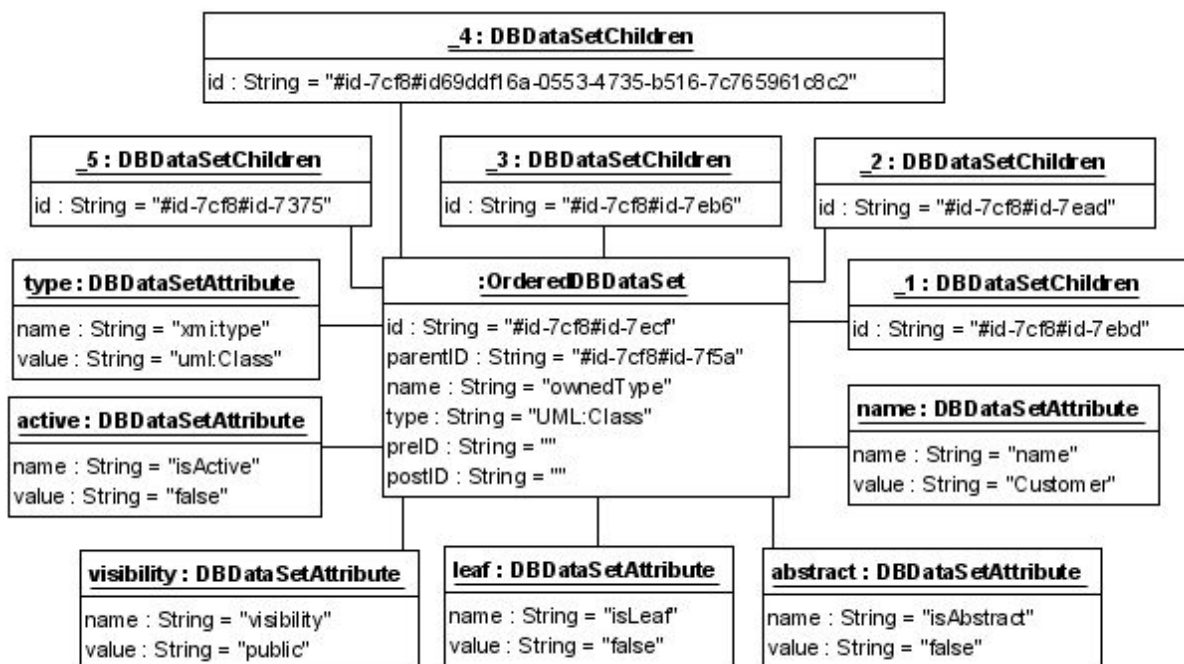


Fig. 6. Object diagram representing class Customer

4. Communication Architecture

The common use of UML models is more efficient if these models are stored in a data base on a computer accessible by all involved users. We realized such an approach using a client/server architecture. As shown in Fig. 7 we use an interface *IDatabaseRW* on the side of the server. The most important operations of the interface are *write*, *insert*, *change*, *read* and *readChild*. These operations are used in order to write objects of the classes *DBDataSet* or *OrderedDBDataSet* to the database or read it in the reverse direction. Further operations like *open*, *close* or *compress* realize the management of the data base. The server interface is realized by the class *DatabaseRW* in Java.

There is an interface *ISerializeDB* on the side of the client. This interface defines operation in order to open or close the database and in order to read or write data. The class *DBClient* is implemented in both Java and C++. *GETECA* uses the C++ client, and the *UML Designer* and the *XMI Converter* the Java client, respectively.

The communication between the client and the server is realized by sockets. The data type used for data transmission is a byte stream. There are two streams: an incoming stream to the database and an outgoing from the database. The incoming stream contains commands for server and necessary parameters. The outgoing stream contains the requested data. The incoming stream uses following format:

$$\langle \text{command} \rangle \langle \text{parameter}_1 \rangle \dots \langle \text{parameter}_N \rangle.$$

The elements in such a command are separated from each other. As a separator sign we use the “?” character, for example *?open?c:\\db\\?*. The command and an associated set of parameters are obligatory elements.

As answer to a command the the outgoing stream, returns is the data using the following format:

$$\langle \text{DBDataSet1} \rangle \dots \langle \text{DBDataSetN} \rangle$$

where the elements *DBDataSet* are separated by “|” characters .

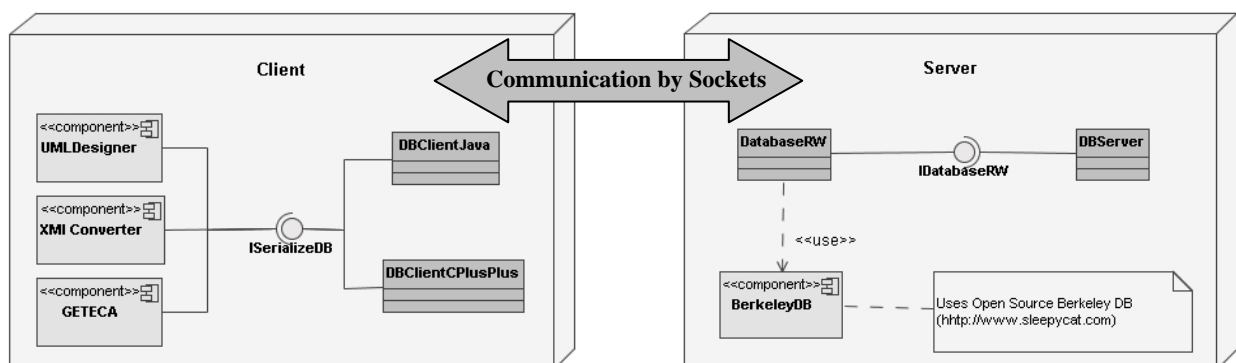


Fig. 7. Communication between client and server

5. Experimental Results

In order to show the advantages of our method of storing of UML models into a central database, we compare the periods of time needed for reading of UML models from BDB including the creation of the internal data structure of CASE tool for different UML models.

Additionally we compare such periods of time for both a local Berkley database and on a Berkley database on a remote server.

Table 2.
Experimental results for some models

#benchmark	# model elements	Reading and parsing time (sec)					
		UML Designer		GETECA		XMI Converter	
		local	remote	local	remote	local	remote
1	1599	13,3	51,6	1,9	4,1	6,9	54,7
2	4669	40,8	131,7	2,1	5,3	25,2	161,1
3	1361	17,1	42,8	2,1	4,1	5,4	47,3

The first column of Table 2 indicates the UML model (benchmark). The second column gives the number of all elements in the model which characterizes the complexity of the UML model. The next columns specify the periods of time for reading and parsing the UML models by three tools in the design chain. For each CASE tool it is given the measured period of time first using a local database, and second using a database on a remote server. The *UML Designer* and the *XMI Converter* must read all model elements and all associated graphic information. The test tool *GETECA* reads the model only a partially.

Conclusion

This paper deals with the data exchange between CASE tools in the design chain. For high level designs of discrete devices UML models gain in importance. The exchange of UML models between CASE tools can be realized by XML files using the XMI 2.0 standard format. The disadvantages of such files are their large size and the time consuming sequential access. Alternatively, we suggest the application of a central database in order to exchange UML models between CASE tools in the design chain. The introduced unique interfaces allow the exchange of models between several CASE tools using different programming languages. Our experimental results show both local database speeds up the access significantly and the comfortable access to a remote database must be paid by a maintainable delay.

References

1. Dorotska, Chr.: Generation of test case templates using the Boolean representation of software model in: Steinbach, B. (Hrsg.): Boolean Problems, Proceedings of the 6th International Workshops on Boolean Problems, 23. - 24. September 2004, Freiberg University of Mining and Technology, Freiberg, 2004, pp. 251 – 256.
2. XMI format specification, <http://www.omg.org/cgi-bin/doc?formal/2005-09-01>
3. Fowler, Martin: UML Distilled. A Brief Guide to the Standard Object Modeling Language. Addison-Wesley Longman, Amsterdam, 2003
4. Meta-modeling technique. <http://en.wikipedia.org/wiki/Meta-Modeling>
5. T. Grose, G. Doney, S. Brodsky, Mastering XMI: Java Programming with XMI, XML, and UML, John Wiley & Sons, OMG Press, 2002.
6. XML Metadata Interchange. <http://en.wikipedia.org/wiki/XMI>
7. Architecture of Berkeley DB. http://en.wikipedia.org/wiki/Berkeley_DB