Information Systems Framework Synthesis on the Base of a Logical Approach

E.A. Cherkashin[*], V.V.Paramonov[*], R.K.Fedorov[*], I.N.Terehin[**], E.I.Pozdnyak[***], D.V.Annenkov[***]

[*] Institute of System Dynamics and Control Theory SB RAS, Irkutsk, Russia

[**] Institute of Mathematics, Economics and Informatics of Irkutsk State University, Irkutsk, Russia

[***] National Research Irkutsk State Technical University, Irkutsk, Russia

{eugeneai, slv, fedorov}@icc.ru, {i.terhin, evgenij.pozdnyak}@gmail.com, annenkov_d@mail.ru

Abstract - We consider an approach to the information system framework synthesis. This approach implements OMG's Model Driven Architecture transformation on the base of combination of logical and imperative programming languages. Information system is modeled using UML Class Diagram. The transformation procedures are represented as rules and source code templates. The generated framework is a set of source code modules, which form libraries for further development. An example of approach application and further improvement of the transformation implementation are considered.

I. INTRODUCTION

Information systems (IS) at present are constructed on the base of a common scheme, where IS consists at least of the following three subsystems:

- *Data Warehouse* (*Storage*) provides persistent data layer for program objects, storage formats, and productive access to the stored data.
- *Application Control Layer*, which usually is referred to as business-logics layer; it is a domain object interaction model implemented as a program. The layer mainly realizes changing the warehouse data, providing the soundness with respect to domain.
- *User Interface* represents stored and processed data for users and propagates events initiated by users to application control layer.

Most of the information systems also have analytical and report generation subsystems.

At present there are popular approaches used in construction of complex IS, rising development performance, namely

- Component architectures and environments, which allow high code reuse,
- Visual modeling of various aspects of the project under development followed by a code generation.

Complex environments such as SAP R/3, JavaBeans, EJB, CORBA, COM/DCOM/ActiveX, .Net are examples of the first approach. Their libraries include professional grade relatively abstract implementations of key subsystems. Developers combine and specify predefined behavior of the library modules to the problem domain. We consider that Rapid Application Development (RAD) systems belong to the component environments. Famous examples of RAD-systems are Borland Delphi/C++ and their contemporary derivatives, as well as Microsoft Visual Studio.

Visual modeling techniques, e.g. Computer Aided Software Engineering (CASE), allow one to deal with complex systems and projects, representing them as an abstract formalized model. CASE-systems use UML to model IS implemented in object-oriented programming and storage environments. CASE instrumental software has code generation routines to convert visual models into source code modules. Usually, the generation routines are mutually independent and represent a viewpoint of CASE-system manufacturer to the process of the visual model representation. There is no standard approach to user interface generation in popular CASE-systems.

Visual modeling is intended for takeover the complexity problem during software design and manufacturing, as well as it is a way of formalized communication between developers and customers. Model Driven Architecture (MDA) [1] is a further development of CASE-system aimed to provide solution for the following problems of IS development:

1. Rapid development of software construction technologies and programming techniques, results in frequent change of software development platforms and accumulation of legacy source and binary code.
2. Necessity to support a number of parallel versions of the software on various hardware platforms and operating systems; for example, most of popular Internet services have applications for mobile platforms (iOS, Android).
3. Reuse of models and corresponding implementation source code in new projects, and accumulation of formalized knowledge on designing and implementation of IS subsystems.

Key concepts of MDA are CIM, PIM, PSM, and PDM. *Computation Independent Model* (CIM) reflects software's external requirements – its interfaces. CIM hides internal structural elements, and therefore can be used to define specifications and checking requirements.

The research is supported by Russian Foundation of Basic Research, grant No 10-07-00051-a.

Platform Independent Model (PIM) is a model of the software reflecting most of the structural and some semantic aspects of the software, but this model contains no information about implementation of the structures on the target program architecture. UML Class Diagram which is extended with some tag values and additional stereotypes is a relatively common example of PIM. The extension (marking) allows one to denote implementation nuances for structures. *Platform Specific Model* (PSM) is a model, which can be implemented as a source code of the subsystems, e.g., it could be a physical structure of a relational database, which is directly (algorithmic or by means of code templates) translated into DDL SQL-requests.

MDA formalizes part of *Software Life Cycle* concept [2], which reflects a path from domain to results of implementation stage (see fig.1). Initially an *idea* of a program is proposed. On the first step basic terms and functionality requirements are iteratively collected, which correspond to MDA's CIM. The next step is a requirements analysis, which results in forming a *general project outline* corresponding to PIM. The developer's design activity results in *detailed system design*, i.e., PSM, followed by the implementation stage of the software components.

MDA is a methodology for developing software by means of partial automatic source code generation of IS from visual models, so it can be considered as an approach to generative programming [3,4]. The main distinction from generic CASE-systems is that the code generation routines are not fixed and can be extended and adapted to the project requirements and developers' way of structures and functions implementation. It can, e.g., be adapted to describe even IS based on component architectures.

The transformation of the PIM into PSMs is carried out under control of a Platform Description Model (PDM). PDM contains information and algorithms of PIM's structure analysis and generation of corresponding data structures in PSMs. Sometimes PSM is understood as specific variant of PIM. The tag values and stereotypes are used to direct the transformation of PIM's structures into certain frames.

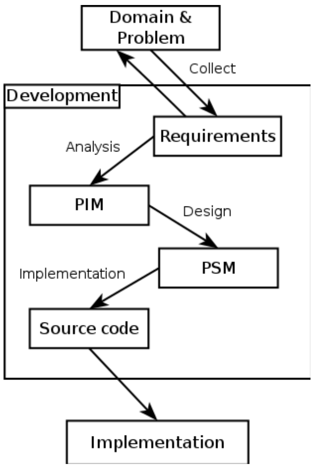

Figure 1. MDA reflects the Software Life Cycle

The aim of our research is to create a complex integrative MDA technology to support designers and programmers with flexible transformation technique, which is by nature adaptive to their peculiar way of software development. Especial interest for us is adaptation of the MDA to extreme and agile programming. This paper devoted to consider our experience of an approach to transformation implementation based on substantial use of a local language and rule-based inference systems.

II. Transformation implementation

There are a number of approaches to the transformation implementation. Algorithmic approach, where all the transformation procedures are implemented with an imperative programming language; XSLT transformations allow to represent transformation as production rules; graph theory and graph transformation; usage of domain specific language [5]. We use logical approach to define transformation as a set of productions and a pattern-directed inference engine [6] and a transformation scenario.

Patterns are represented as a mix of Prolog and Python programs. Pattern query is a Prolog rule located in so called *__doc__*-strings of Python instance methods. The bodies of instance methods are execution parts of the patterns. Parameters passed to the methods are results of corresponding pattern queries inference. Instances itself are modules, i.e. a set of patterns and algorithms that transform part of PIM into a part of PSM. Consider the following example of patterns which are supposed to recognize and generate SQL query to create relational database table for storing object instances of IS under development.

```
class RulesMixing: # Set of patterns
    def rule_primitive_class(self, cls, oid, oidType):
       # figure out the basics of relation coding
       # in relational tables
       """ % this starts __doc__ string
       primitive_class(Cls, OIDAttr, OidType):-
           element(Cls, 'Class'),
           \+stereotype(Cls, 'abstract'),
           stereotype(Cls, 'OODB'),
           \+internal_only(Cls),
           stereotype(Cls, 'primitive'),
           attribute(Cls, OIDAttr),
           type(OIDAttr, OidType),
           stereotype(OIDAttr, 'OIDkey'),!.
       """ # this ends __doc__ string
       self.BASE_CLASS = cls
       # we found the root of the class hierarchy
       self.BASE_CLASS_NAME = self.getName(cls)
       self.OID_NAME= self.getName(oid)
       # attribute for object reference
       self.OID_TYPE = self.coerceAttrType(oid, oidType)
       # choose a type for the object reference
       return self.BASE_CLASS_NAME, cls, oid,
              self.OID_NAME
       # the values are passed to a for statement.
    def rule_persistent_class(self, cls):
       # a class is persistent if its instances
       # are to be stored and it is not a type.
       """
       persistent_class(Cls):-
           element(Cls, 'Class'),
           \+stereotype(Cls, 'abstract'),
           stereotype(Cls, 'OODB'),\+internal_only(Cls).
       """
       # this pattern has no body
 . . . . . . . . . . . . . . . .
class SQLTranslator(Translator, RulesMixing): # a module
   # it generates SQL-script of database structure
   def genClass(self, cls):
       # Generate SQL-script for a class cls
       answer = [] # list of source lines
```

```
if cls in self.generated:  # is it already
    return answer          # generated?
for _, parent in self.query('class_parent',
    (cls, '#')): # generate all ancestors
. . . . . . . . . . . . . . . . . . . . . . .
name=self.getName(cls)     # name of the class
doc=cls.getDocumentation() # documentation
if doc:                    # is it not empty?
    answer.append('/*\n%s\n*/' % doc)
attribs = self.genSchema(cls)
# generate table attributes
if not self.isEmpty(attribs):
    answer.append("CREATE TABLE %s (" % name)
    answer.append(attribs) # table attributes
    answer.append(")%s;" %
        self.getTableType(cls))
else:
    print "The class has no attributes."
self.addFact("oodb_table('%s', '%s')" %
    (cls.getId(), name)) # assert conclusion
# on a relation of class to table
self.generated.append(cls)
return answer # return generated code
```

Method *genClass* is executed from outside for each answer *Cls* of *persistent_class(Cls)* query. Structure of the base class, recognized by rule *rule_primitive_class*, greatly affects a way of object references representation of the rest of the relational database tables.

This approach has cumulative advantage over above mentioned techniques: expressive production-like transformation representation, powerful imperative and retrospection abilities of Python, existing template engines used in Python web frameworks, and it is a tool, which is not tied to specific set of development environments.

We develop a software designing technique for MDA based on multistage transformation of PIM into a PSM consisting of specific submodels (fig. 1). To transform the UML models its XMI (XML Metadata Interchange) file is loaded. This format is a kind of XML, so DOM2 API is used to access PIM's structure. XMI is a standard data format supported by various proprietary and free software technologies and libraries. The DOM2 tree is translated into Prolog facts by means of requests from patterns. Object Constraint Language (OCL) expressions are extracted from PIM and represented as syntax trees.

At first a general reasoning about object structure is carried out, basic properties are recognized. Other modules use the reasoning results to refine implementation variants of synthesized program objects.

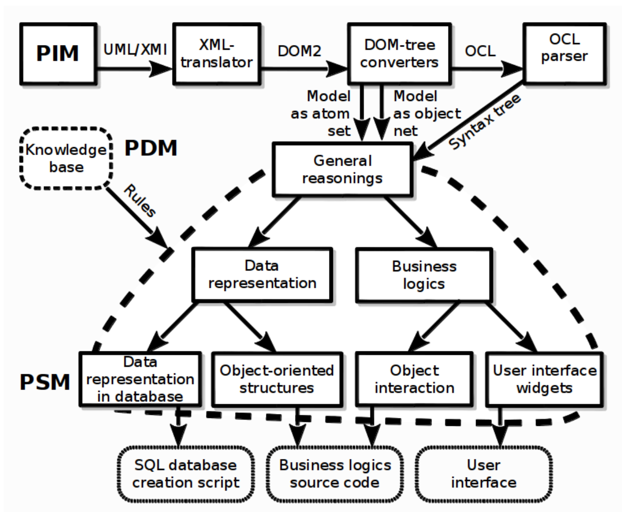

Figure 2. Architecture of transformation engine

Each module specifies PIM's structures with additional facts about existing structures and creates new objects and relations. For example, SQL database transformation module merges inherited abstract part of attributes to the class and generates table description on the base of the merge. The process is controlled by scenario represented as a list of leaf nodes to be executed. If all nodes of the scenario are executed and all their solutions (queries) are satisfied and processed, then the set of all the facts in working memory defines PSM. The source code is generated on the base of obtained PSM. A generator module executes a query and fills in a source code template with query results.

Results of transformations and code generation are combined in object libraries. Objects from libraries are used in construction of business-logics of developed IS. Programmers supplement generated code by inheriting it in new classes. This approach partially solves the problem of generated source code modification by programmers.

The XMI file can contain not only the structure information, but also some semantic values for its elements. This information used to increase the control over the transformation procedure, in particular, for filtering information by using some criteria. UML have the following semantic definition language structures:

1. Stereotypes to create new elements of UML;
2. Tagged values to create properties for the elements;
3. Constraints to formally define logical constraints, invariant, pre- and postconditions for a method invocation.

Let us consider a simple example. Assume that there exists a class in an UML Class Diagram that has at least a string field *name*. If we mark the class with, e.g., «Reference Book» stereotype, then all many-to-one relations to the class can be interpreted in relational database context as many-to-one relation and corresponding tables, and reference fields *ID* are generated. Having recognized the stereotype and the relation, user interface generator can construct a widget and its controller (in sense of Model-View-Controller paradigm) to select appropriate record from the reference book and store the reference book *ID* in corresponding table and object. Now, the generated code in various subsystems is logically and mutually depended.

In order to adopt the transformation engine and its knowledge base to developer's instrumental software and technologies, one imports Python module, inherits and modifies its set of patterns and generation modules, specifies new module in scenario. In the application example in the following section we used inheritance to refine a generic SQL relational table transformation to specific properties of MySQL server.

III. APPLICATION OF THE TECHNIQUE TO CONSTRUCT A FRAMEWORK OF AN INFORMATION SYSTEM IN MEDICINE

In 2005 we applied our transformation engine in the life cycle of medical IS named "Population cancer registry" development for recording cases of cancer incidence in Irkutsk Regional Oncology Center (hereafter hospital). The IS was to accumulate data on the cases happened in Irkutsk Region (Irkutsk Oblast). The territory is about 768 000 km^2, and its population is about 2 500 000 people (2011); every year around 8000-9000 cases are recorded. The necessity of the development is dictated by Ministry of Health and Social Development of the Russian Federation by a corresponding directive in 1999.

Previous version of the IS was based on Microsoft Access'95 and designed as stand-alone applications with common server database developed since 1999. As in 2005 there were no high bandwidth channels to the hospitals subordinate clinics and oncology medical offices, as well as most of the offices were not connected to Internet at all, the input data came to the clinic as filled in printed forms by regular mail or with courier every month. Then all the forms were recorded into database by stuff of organizational-methodical department of hospital. The IS and operating system peculiarities did not allow the usage of the system for medical doctors directly in their offices: response time of IS was very low and there were no obvious ways of overcoming that problem; the system had also closed proprietary design.

In that time Russian government began to support a number of programs of development, including digital medicine and communication channels quality and productivity improvement. In the same time Internet technologies and software as a service started to dominate on market; a number of open-source technologies become mature. Hospital and regional administration decided to realize new version of IS, which are to be the international platform of network infrastructure accumulating all oncology data streams from various medical information systems in the region. There was also financial support in amount of 10 400 euro for the initial state of the project.

Initial condition to the IS development was somewhat indefinite: there were a diversity of hardware and software (out-of-date personal computers and operating systems); structures of input documents though were approved by the above mentioned decree, but they were informal; there was a lack input data to required report forms; also there was no standard strategy of IS implementation as an application for user. In this situation we decided to organize development mostly on an abstract level, which would allow generating frameworks for randomly appeared new requirements.

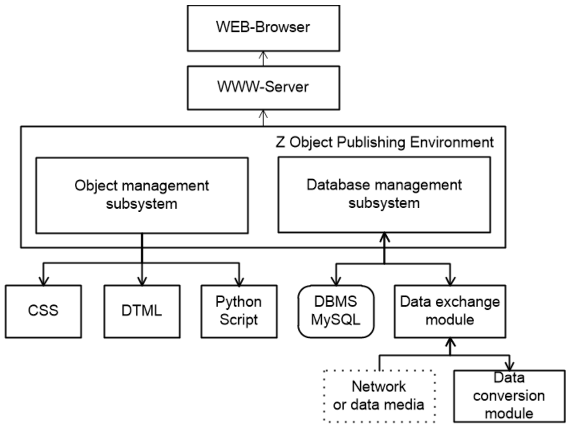

Figure 3. Architecture of Population cancer register medical IS

As the UML editing tool we used Gentleware Poseidon for UML Community Edition v. 3. The IS's PIM was presented as the marked UML Class Diagram and contained more than 100 classes, interfaces and other auxiliary structures. PIM represented whole class/instance, records and enumeration structures. Most of the classes were marked by «OODB» stereotype to be stored in relational database MySQL-4.1 as objects. Some of the classes were marked as «Reference book». Records represented joined lists of attributes, which were included in class as a complex structural attribute (e.g. passport data), but stored in the same table as their class. Enumerations are structures, whose attributes used as constants in database, business logics code and user interface.

Attributes of classes were marked with various tag values. For example, tag value "name" denote a notation of the attribute in user interface forms, "index" (true or false) suggested to the transformation engine to add an index in database definition for the attribute; "index_kind" (btree, hash, etc.) refined the variant of index engine; set of "widget:…" tag values controlled variants of attribute user interface representation. By means of the tag values we denoted the storing engines for persistent classes, grouping attributes on user interface forms, the layout was implemented manually.

As target platform the following software were chosen: Gentoo Linux OS; MySQL-4.1/5.1 with InnoDB and MySAM engines as relational database server; object-oriented Internet application framework Zope-2.7.3 and Python programming language as business-logic implementation environment; XML as data presentation format. Transformation engine has about ten modules; each module has about ten original (noninherited) rules. In fig. 3 architecture of the IS is presented.

The constructed transformation system has generated a complete DDL script for database, representing all the classes as objects referring each other through object identifier OID and some of retrospection instance data (inheritance between classes); complete set of business-logics Zope objects represented as Zope folders with full support of contexts; complete flexible templates for object-relational layer between MySQL tables and Zope objects, the layer engine based on Zope SQL Methods; set of markup Zope Page Templates for the presentation format for export/import objects; template of Pascal language program for data export from previous version of IS; C-language efficient data importer from XML-representation; templates of input fields for user interface. For Zope objects we have also generated methods reflecting class-to-class relations, e.g., methods to get all tumor cases for a patient. The transformation engine has also generated the metadata, which used by Zope methods for special utility purposes. Our instrumental software has generated 91 tables for database and more than 8000 lines of source code. The kernel of the transformation engine and most of the rules was implemented in the context of the project for 3 human-months. The transformation cycles took about 1 minute.

The generated source code, methods and modules were integrated into IS with calling the code or inheriting it, so we could regenerate the framework without loss of later made changes of source code. Database data integrity supported manually and was periodically reimported from the database of the previous version. User interface was constructed manually from widgets, supplied by object's methods.

Using the MDA in this project we faced a number of problems. To the end of the design stage PIM occupied 4 m^2 and could be displayed only partially on the screen, some time was spent for periodical layout adjustment (location and color) of the model classes by their properties. Memory integrated YAP (Prolog) and Python transformation engine started to crash on big-size input PIM, we switched to less productive process- and stream-integrated version of the engine. Practictitioner programmers did not share our optimism about MDA usage: they preferred verbal form of modeling; they were forced to implement report generation subsystem. Some years later Gentleware decided to charge for new and all old versions of Poseidon for UML CE, but none of investigated free UML editors can load our model now.

At the end of the design stage we had almost functional IS supporting most of required function but it have somewhat ugly interface: medical stuff was too busy to help us with testing and refining. Presently the PIM is used on-site programmers as source information for recreation of production grade version on Django/PostgreSQL platform. Now the IS is used in the oncology hospital.

IV. Current development of the technique

Further development of the technique is aimed at raising performance and expressive abilities of the transformation system, as well as its reliability. Main problem is that it is hard to provide efficient and in the same time sound integration of Prolog and Python: both systems have its memory management units but different memory management strategies. We decided to shift the system to use powerful expressive abilities of LogTalk Prolog [7] macro package supplying it with an imperative subsystem and a template engine. LogTalk is an object-oriented logic programming language that can use most Prolog implementations as a back-end compiler and inference engine. As a multi-paradigm object-oriented language, it includes support for prototype and class inheritance, protocols/interfaces descriptions, component-based programming through category-based composition, event-driven programming, and high-level multi-threading programming.

In the new implementation [8] of the transformation we take advantage of the same object-oriented hierarchical modular architecture as before, but set up another goal - to support transformation in both directions: from abstract PIM to source code and from the source code to abstract models. This should partially allow developers to

- modify generated source code and conserve changes, i.e., account them in PSM and PIM;
- develop software on the various levels of abstraction;
- accumulate libraries of complexes of models and their implementations as well as transformation modules.

The results of the investigation will be realized in a software development environment, integrating UML design software and source code IDEs. The integration engine is to be based on change propagation [9]: the modifications made are recognized by the environment and pushed to other utilities.

A. *LogTalk transformation module example*

Let's consider a code example written in the LogTalk programming language. We have the following interface class for access to the loaded XMI DOM2 tree. This class is used in Model class, that is a primarily recognize the model in XMI file.

```
:- object(class, instantiates(class)).
        :- private(attributes_list/1,
                operations_list/1,
                . . . . . . . . .
                parse_operations/2).
        :- public(new/5,
                new/2,
                name/1,
                operations/1,
                attributes/1).
. . . . . . . . . . . . . . . . . .
:- end_object
```

Next code is an example of a transformation procedure

```
:- object(TransformToSQL(Class, Type) )
. . . . . . . . . . . . . . . . . .
%Parametric object allows to customize some parameters
        database(Database) :-
                parameter(1, Database).
        class(Class) :-
                parameter(2, Class).
        databasetype(DatabaseType) :-
                parameter(3, DatabaseType).

        gen_sql(Name, Attributes, Output) :-
                append("CREATE TABLE", class::Name,
                        Output),
                append(Output, ::gen_attributes(
                        class::attributes,
                        Attributes), Output),
        append(Output, ::databasetype, Output).
. . . . . . . . . . . . . . . . . .
:-end_object
```

Call to the *gen_sql* method looks like *TransformToSQL(Student, "Inno DB").*

B. *Change Propagation*

Definition of change propagation in [9] from the MDA pint of view can be interpreted as a way of transformation. Transformation engine compares two versions of PIMs, recognizes the difference and corrects PSM and source code in the corresponding points of change. The transformation approach uses specially stored links between objects from PIM and corresponding generated object in PSM. When an object changes or deleted its image in PSM traced by its link.

We suggest extending this idea to allow the propagation in both directions, including from PSM to PIM. This should results in the following additional advantages:

- Record the stages of the development as complexes of abstract models and corresponding source code fragments;
- Allow designers to transfer the model complexes between projects.

To deal with stated extension and planned features we proposed to apply the theory of systems of complexes (configurations), successfully used in geography research [10], to software life cycle, implying that the software development is a natural process. The theory shows how to represent model elements, relations between elements, transformations and links as complexes, as an element of a category. In [10] we shown that as the model complex and a change are elements of the same set of complexes, hence the change is expressed with the same language structures as the model complex. Similar arguments are true for transformation modules; they can transform both the model and its change.

Some practical examples have been found, like *diffutils* package of any Unix distribution, which demonstrate the investigation results. The package allows programmer figuring out the differences between two versions of a source code text file (*patch file*) and apply the *patch file* to the sources. The package contains two main utilities *diff* and *patch*. The first utility compares two ASCII or Unicode texts and produces a new text file with a representation of the difference between input files.

```
--- t1.cpp 2012-02-12 11:50:42.668030039 +0900
+++ t2.cpp 2012-02-12 11:52:44.944957992 +0900
@@ -2,6 +2,7 @@
     char * name;
 };

-class Pupil:Person {
-    Class * cls;
+class Student:Person {
+    char * stud_no;
+    Group * group;
 }
```

In this example shows that the difference is shown with the same ASCII or Unicode characters: the text is shifted by one character right so the first column became control column, where characters define a modification. The space character denotes a context of the text part under change. The minus character denotes part that to be removed, and plus character denotes the new text composition to be added into the context. As in one patch file the whole project change set can be stored, special format substrings "---" and "+++" are used to denote the particular files, and "@@" is used to denote general relative position of the first line of the context in the files.

Links from PIM to PSM express context of changes more precisely than patch file format, and allow back transformation engine to recognize the differences. We suggest using an intermediate text format between PSM and generated source code. This format reflects the links and also account that the generated source code in a general case is not a flat text anymore. The format can be constructed on the base of a D.Knuth's Literate programming [11] implementation. Literate program itself is a tagged tree representation of computer program, and those tags can be generated from PIM and PSM and reflect the links and abstract objects of the model. In opposite direction the changes of the source code now can be more precisely recognized having the tags at hand. There are well developed tangling and untangling algorithms (reverse direction), e.g. in [12].

V. Conclusion

We have considered an existing implementation of transformation engine for Model Driven Architecture approach to software development in the case of information systems. The approach is based on mixing high level object-oriented programming language Python and logic language Prolog. The transformation is represented as network of modules. Each module carries on a subtransformation and implemented in a pattern-directed fashion. An example of application in medicine and further development of the system is considered.

One of the aims of the research is to construct a software development tools based on analogy. For example, having stores in a revision control systems all the states, models and stages of MDA software development as change complexes, it probably be possible to construct new sequence of differences for new original model.

References

[1] D.Frankel "Model driven architecture : applying MDA to enterprise computing". - New York: Wiley, 2003.

[2] Si Alhir. Understanding the Model Driven Architecture (MDA) URL: http://www.methodsandtools.com/archive/archive.php?id=5. (access date-29.09.2012)

[3] M.M.Gorbunov-Posadov, "The way to grow a program", Open Systems Journal, 2000, N. 10, pp. 43–47. (in Russian) [English version URL: http://www.keldysh.ru/pages/softness/growE.htm (access date-29.09.2012)]

[4] K.Czarnecki, U.Eisenecker. Generative Programming: Methods, Tools, and Applications, Addison-Wesley, Reading, MA, USA, 2000. 864 p.

[5] M.B.Kuznetsov. "UML Model Transformation and its Applications to MDA Technology", Programming and Computer Software, Russia, Vol.33, 2007, pp. 44-53.

[6] I.Bratko. "Prolog Programming for Artificial Intelligence". Addison-Wesley pub. co. 1986, 423 p.

[7] P.Moura "Logtalk: Design of an Object-Oriented Logic Programming Language". PhD thesis. Universidade da Beira Interior, 2003. [See also, "Logtalk. Open source object-oriented logic programming language". URL: http://logtalk.org/ (access date-29.09.2012)]

[8] E. A. Cherkashin, S. A. Ipatov. "Logical Approach to UML-model processing of Informational Systems". Journal of Conterporary Techologies. System Analysis. Modelling. 2009. N 3 (23). pp. 91–97. (in Russian)

[9] M. Alanen, I. Porres, "Change Propagation in a Model-Driven Development Tool," in Presented at WiSME part of UML 2004, 2004.

[10] E.A.Cherkashin, V.V.Paramonov, et al, "Model Driven Architecture is a Complex System", E-Society Journal Research and Applications. Volume 2, Number 2, 2011, pp. 15-23.

[11] D.E.Knuth. "Literate programming". The Computer Journal, 27(2), 1984, pp. 97-111.

E.K.Ream. "Leo's Home Page" URL: http://webpages.charter.net/edreamleo/front.html (access date-29.09.2012)