

Model Driven Architecture is a Complex System

E. A. Cherkashin*, V. V. Paramonov*, S. A. Ipatov*, V. S. Tertychniy** and I. N. Terehin***

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

** Irkutsk State Technical University, Irkutsk, Russia

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

eugeneai@icc.ru

Abstract – An abstract formalization of the software development life cycle (process) in the theory of complex systems and complexes is considered. The formalization highly depends on so called reference set, which is a basis of bundle of the life cycle into a set of structures, e.g., various software model representations. An example, which appears to be a generalization of Model driven architecture (MDA), is considered, as well as the present approaches and technologies for the software development if the proposed model implied.

I. INTRODUCTION

Any software development life cycle consists of distinct stages and involves various agents (manages, software developers, users, etc.) and technologies, such as mathematical modeling, information representation modeling, information processing and visualization, user interfaces. At the very beginning the problem stated represents an ideal object, which is specified during the stages as various linguistic, mathematical and information models. At the stages of implementation the models are represented as algorithms and data structures, which are realized as program objects and components. At the testing and deployment stages the software is used by testers and users. In the general case new ideas and problems affect the life cycle at any stage. For example, new requirements are analyzed on the basis of the obtained experience and new software life cycle is constructed; new implementation technology implies reconstruction (translation) of the source code into new language and corresponding data structures adaptation; the user's suggestions of the user interface modifications imply data structure and source code reconstruction.

Various combinations of stages form a number of life cycle schemes, such as waterfall and spiral models, 'V'-model, agile and extreme approaches, iterative and incremental development, and various improvement models [1]. All the approaches use models of various degrees of abstraction and formalization. We consider a general case of life cycle as a process of adaptation of new ideas, requirements and specifications implies modification of all the models (formalized and implied). Thus, the software development process is represented as propagation of the modifications.

The problem we consider in the paper is to construct an approach to describe the process of the modification

propagation as a basis of a corresponding instrumental environment. The necessity of the modification propagation results from application of the theory of complex systems and complexes to software life cycle.

II. THE THEORY OF COMPLEX SYSTEMS AND COMPLEXES [2]

Complexes (compositions) and systems of compositions (configurations) X_i are formed from combinations of various elements, components, systems, complexes, systems-complexes. Configurations X_i are different kinds of complex system's polymorphism. The universal structure X contains all X_i , and in this case it is analogous to notion of set of all subsets or category of all the subcategories. The connections between compositions are represented by means of mapping (morphism) $F_{ij} : X_i \rightarrow X_j$. The composition connected with morphisms (represented a category in the mathematical sense) form a complex. The comparison of the two compositions X_i and X_j , namely $\Delta X_{ij} = X_j \setminus X_i$, shows the dissimilarity between j -th composition and i -th one. For example, let X_i , X_j be different UML models of software before and after adding a new structure, so ΔX_{ij} reflects a complex of the current development step. If X_i , X_j are two UML models of different software systems, then ΔX_{ij} fixes, in particular, their structural dissimilarity, and F_{ij} is a comparison relation. The structure ΔX_{ij} is also a composition and belongs to the set of all the comparisons ΔX . The set F is a set of all possible mappings F_{ij} .

There are also a reference set of comparison I ; it is the interval $[0, 1]$, which is a metric linearly ordered inductive continuous bounded above and below set of points. The one-to-one correspondence of compositions and reference set points is denoted by relation $(\leftrightarrow)$, for example, $I \leftrightarrow I$ means that each point I turns into itself; $X \leftrightarrow I$ means that any composition from set X is one-to-one corresponded to a definite point of the interval $[0, 1]$. In the latter case I is a bundle (fibration) basis of X , in other words the differentiation X onto compositions is

conducted by means of comparing them with points (numbers) from the interval $[0, 1]$.

The axioms of complex systems' theory are to compare compositions and their connection functions with an identification index of an order and together.

$$1) X \leftrightarrow I; 2) F \leftrightarrow I; 3) \Delta X_{ij} \leftrightarrow F_{ij}. \quad (1)$$

The axioms 1 and 2 transfer all the order properties of the set I to the sets of all compositions $X_i \subset X$, their comparisons $\Delta X_i \subset X$, and mappings $F_{ij} \subset F$. This also implies that all the combinations are toposes, i.e. they are linearly ordered structures with the individual measure from I . By means of I the structures, their comparisons and mappings are one-to-one connected to each other.

$$X \leftrightarrow F \leftrightarrow \Delta X \leftrightarrow I.$$

This emphasize a necessary moment of complexing, the complexes, their comparisons and mappings are equivalent (identical in the sense of dialectical logics), and have the single identification index in I . The complexes are self developing systems, where each alteration is based on its structure $\Delta X \leftrightarrow X$. Complexes are static formations, the transitional states with partially formed connections cannot be related to them.

Complexes are linear sequences of morphisms (categories)

$$\dots \rightarrow X_i \rightarrow X_j \rightarrow X_k \rightarrow \dots;$$

the contained compositions form homologous series of comparison, as each composition has a measure from I , this series is also homotopic. Homologo- homotopic series form comparisons and mappings. All the elements of the series are functionally similar, that's why they can be considered as series of analogs.

Any fragment of a homological series $X_i \rightarrow X_j$ is a complex, which corresponds to a complex of comparison of the dissimilarities $\Delta X_i \rightarrow \Delta X_j$. Consequently, a similarity of the structure implies similarity of modifications and vice-versa. Therefore, observed similarity in the structure must be caused by similarities of the processes and their models. The search of structural and dynamical similarity is the main subject of the theory of complexes.

A. Application to Software Life Cycle

In the field of software development the theory could be implemented in various aspects. If we choose the reference set I as detail level of software model representation, then 0 could mean a completely abstract level (just an idea), and 1 denotes completely realized software complex. Further, X_i will correspond to informational models of various abstraction. For example, X_0 is the original idea, X_1 is a textual representation of the requirements, X_2 is an UML model, ..., and at the end of the interval, e.g., X_9 is a completely realized software system. The configurations $X_0, X_1, \dots, X_9$

correspond to points $i_0 = 0, i_1, \dots, i_9 = 1$, (for each $i_k < i_{k+1}, k = 0, 1, \dots, 9$) of the interval $[0, 1]$.

The idea X_0 is explained by means of its terminological basis of formalization, and references to the software domain and problem stated. The reference could be an existing ontology or a textual representation of the problem. The dissimilarity ΔX_{01} denotes the additional information about requirements to the software under development. This is a result of the problem and domain decomposition. As X_1 one can understand as IDEF0 model extended with formal or informal specification of its structural elements. The morphism F_{01} is a creative function done by system analyst under restriction of some technique, e.g., SADT [3].

The dissimilarity ΔX_{12} denotes the result of system designer's activity F_{12} of conversion IDEF0 model with the requirements into set of UML diagrams. All ΔX_{ij} have their corresponding interpretations.

Theory's axiomatic basis here is realized as follows:

- Axiom 1 in (1) denotes, that any software system can be considered (modeled) at various abstraction levels denoted by I , and the inverse direction means, that for any set of models the bundle basis I could be constructed;
- Axiom 2 means that it is possible to develop software as model transformations and refinements, as well as having developed a sound in some sense software, then it has sound model set representation (formalized or just implied);
- Axiom 3 for each model specialization or transformation a set of methods (techniques, tools) to carry out the development could be found or developed, as well as methods and instrumental software used to develop software by means transformation of corresponding models.

The interpretation of the identity $X \leftrightarrow \Delta X$ means that the process of software development is based on its structure, ΔX results from testing and exploitation, and the software development is an improvement of the set of the models. The complex in this example also form a homologous (homotopic, analog) series, in other words, one can make advantage of the same steps as in an early project using the same set of models X to develop new software.

There are various approaches to automatic transformation of the models in the field of software development, such as IBM Rational Unified Process, Model Driven Architecture (MDA) [4] and formal methods [5]. The approaches are aimed at automation of creative activity of designers and programmers and implemented in instrumental software. The software development tools having a model at input generate a model or source code, which also we consider as a model. Most of the transformations are formal and deductive; the MDA approach requires a Platform Model and a scenario to specify a variant of the transformation.

A translation of the properties of the theory of complexes to the processes of MDA-transformations results in following conclusions.

- In developing software any state of X , including ΔX should be stored for later usage in transformation
- The stored results of the transformations should be analyzed to extract new knowledge about the transformation specific of the current task and also general templates about designing software.
- Complex ΔX is of especial interest and subject of analysis. Namely, as transformation $X_i \rightarrow X_j$ corresponds to the dissimilarities $\Delta X_i \rightarrow \Delta X_j$, then the instrumental software should correct all the models in X as soon as some ΔX_{ij} has been fixed by a developer.

Thus, it is of purpose to construct instrumental software based on analysis of analogy and propagation of the modifications ΔX .

B. An Example Project

To investigate the possibilities of the software development approach we started a pilot project of notarial office automation. The task selected among stated as the development of the software is deeply depended on users' desire to change: creating instances of documents, correct errors, forming new document classes, composing new document workflows, as well as refinement of user interface aimed at raising the office productivity.

From the functional point of view the notarial office is primarily an organization for document preparation, storage, and retrieval; tracking of the individual's data is the secondary aim that allows producing the document more agile. There exist four user roles in the office; they are "secretary", who fills in the templates, "template modifier", who is an experienced user allowed to construct forms and describe new regular structures found in documents, "programmer", who understands information modeling and implements the routine tasks as program modules, and "notary", who validates and signs the documents. As usual the roles define the set of activities and responsibilities for corresponding users.

During the exploitation of the information system secretaries gain experience, and can evolve in template modifiers. Shifting a user from first role to another can be done as a result of his/her qualification assessment. The assessment can be performed by notary and programmer or by means of testing, for example, answering a set of tests and/or doing test exercises.

Each instantiation of a template can be considered as a copying a document in the storage and its refill with new data or even just an edition of the copy. Modification of the second kind can be interpreted in several ways. Firstly, as mentioned before, it is just refill of the template. Secondly, it is a template body text error correction or a further improvement of the document. Thirdly, the modification could touch the form structure giving rise a new template of existing kind (class), or form even a new

class of templates, e.g., fixing some parameter or substructure.

The modification propagation process is based on a number of document models. Secretaries will fill in documents as HTML forms and edit the generated instances in a WYSIWIG HTML editor. HTML is widely used and support necessary level of the notarial document representation. There are many useful methods of HTML generation and modifications. The difference between documents of various versions is to be propagated to other models. Such models represent the document layout and presentation (CSS), structure of the document class (what parts should be presented in the document and in which order), structure of the form of the template, fixed data structures stored in a rational database, and so on. The structural models are based on corresponding ontologies, e.g., ontology of structure elements of a document, ontology for expression individuals' data, etc.

The system should control the process of the document designing in a dialog with user, acquiring the additional information on user's intention and acting in concordance. Complex problems are described by users as texts for programmers, who implements new features of the software after confirmation of the requirements.

III. IMPLEMENTATION TECHNOLOGIES

Since 2001 OMG exploits Model Driven Architecture (MDA) of software development. MDA [4] is a part of the model considered in the section 1.A. MDA exploits three levels of abstractions to represent software: CIM, PIM and PSM.

The Computation Independent Model (CIM) reflects software's external requirements – its interfaces. CIM hides structural elements, and can be used for define specifications and checking requirements.

The software designing technique of MDA is based on multistage transformation of Platform Independent Model (PIM) into a number of Platform Specific Models (PSM). PIM is a model of the software reflecting most of the structural and some semantic aspects of the software, but the model contains no information about implementation of the structures on the target program architecture. UML Class Diagram extended with some tag values and additional stereotypes is an example of PIM. The extension allows one to denote implementation hints for structures. PSM is a model, which can be implemented as source code of the subsystems, e.g., it could be a physical structure of a rational database, which is directly (deductively or by means of code templates) translated into DDL SQL-requests.

The transformation of the PIM into PSMs is carried out under control of a Platform Model (PM) and a transformation scenario. PM contains information and algorithms of PIM's structure analysis and generation of corresponding structures in PSMs. Sometimes PSM is understood as specified variant of PIM. The tag values and stereotypes are used to direct the transformation of a structure into desired frame.

PMs in most of commercial MDA systems have been implemented on the basis of algorithmic approach. They

are not far from CASE systems translating UML diagrams into a source code by various plug-ins. The main idea of MDA is to allow developer to modify PM according his/her preferences and task properties. Our experience shows that usage of present logical languages and PMs based on formalized knowledge [6] allows us to affect the transformation in an efficient way by means of changing a rule set content.

We use [6] a logical approach to implement transformation. The source PIM is represented as XMI-file version 1.2. As it is a variant of XML, the file is parsed by means of libxml XML parsers into a tree. The tree encapsulated inside a LogTalk module, which processes queries to PIM structure. The transformation procedures and PM is represented as set of LogTalk modules connected with messages. Each module contains a knowledge base to recognize an aspect in the PIM and its derivate structures. The results of recognition form new facts about PSM. The transformation scenario is a set (sequence) of the leave modules, which generate source code and other data structures.

Thus, the generated PSM is represented as set of facts consisting of the subset describing the original PIM, which is obtained while querying the XML tree, and the subset describing the implementation aspects of the software under development. The resulting source code is generated by leave modules by means of templates, so the templates play the similar role as CSS in web, it represents PSM as texts of source codes.

Main advantages of MDA usage in the software development are as follows:

1. Design stage independence of the implementation platform; capability to replace the platform without redesigning PIM.
2. Formal definition of PM: programmers' knowledge is represented as rules and algorithms.
3. Raising the automation level of the life cycle: early stage modifications (design stages) are less expensive to implement in PSMs.

MDA is a great approach and successfully used in development complex software, but it has significant disadvantage, which we are to overcome:

1. Using the MDA in simple projects usually extends time of software construction, although obtained formal PIM and PM models when analyzed could be used in other projects;
2. Currently MDA is of little use in already constructed and implemented systems and systems based on stored data manipulation, e.g., existing informational systems, as modification of information data model results in database structure modification like adaptation to new data structures;
3. Modification of PIM and source code is ignored by the procedures of transformations.

The support of the above mentioned propagation of dissimilarities ΔX_{ij} and modeling whole life cycle's

homology should overcome the disadvantages, and it means, in particular, that the instrumentation software should support the transformation in both directions.

IV. SOURCES OF MODIFICATIONS

When a MDA tool generates a source code, the problem appears when the generated code was modified by a programmer. The modification can be easily lost because of likely following regeneration. One of the ways to conserve modifications is to represent the generated software framework as a library and allow programmers to inherit the code. Changing sources is useful because of programmer can more comfortable figure out the correct data types and names the entities "in place", adjust the procedures to improve performance.

Controlling changes in source code can be realized through using version control systems that can efficiently compare the source code versions, and through developing compilers, which could be aware of the PSM and PIM existence. In the simplest case the difference of the versions is stored as a patch; the patch reapplied each time after the source code regeneration; conflicts are resolved interactively by programmer. Another way is to analyze both versions of the source code as parsing trees, the difference propagated into a PSMs' and PIM's versions. The propagation should be made under programmer's supervision: programmer must supply the information on the meaning of the difference.

To support the propagation on the level of the source code one can take advantages of the literate programming tools and data formats, which can be thought of as a way of hypertext markup of the source code generated from PSMs. Literate programming is a way of source code construction, where the programmer mixes a task description and the task solution – the program – in the same source text file or a tree structure. The program is also constructed from structural parts. The literate programming transformational tools analyze the source structure and generate source code tagged as special cases of comments of the generated program to reconstruct the original structure in case of generated source modification. Some literate programming tools can generate a whole project from one tree (see for example Leo editor [7]) and track some source code modifications.

In MDA case the source structure is a PIM, transformation modules include data about original structure of the PIM as tags into comments of the generated source code. These tags are semantic marks of the source code intervals. In this case the difference of the source codes can be directly associated to the structural element of the source model.

The theory of complex systems states that $\Delta X \leftrightarrow X$. In our case this means, that the structures for the models representation can be used to represent the modifications, also the algorithms of transformations of the models can be used to transform of the modifications. The modifications can be represented in the similar way as patch files as groups <removed substructure/context of removal, added substructure>.

Another way of obtaining new information for models is the texts related to the software domain. New notions could be extracted by means of text analysis of new requirements, as the texts are based on the steady (in time) terminological basis, allowing human beings to understand each other. Texts contain artifacts referencing informational structures of the software, e.g., template word sequences denoting concrete user interface or data structures. There are approaches constructing formal taxonomies (ontologies) from analysis of appearance frequency of terms, see, e.g., [8]. The requirements contain both new restriction and new terms, which possible be a new classes or instances. Two versions of the ontology compared and the difference - new notions and classes shifting in the hierarchy - will reflect the new requirements.

Let us briefly consider a technique [8] for text analysis and thesaurus extraction. The technique's input is a set of texts and output is a thesaurus, where for all terms a subset of the source text set corresponding to the term is associated. The technique consists of four steps:

1. Construction of the stemmed word index of the texts' set.
2. Form a terminological basis as a set of terms; the terms are represented as a sequence of adjacent stemmed words.
3. Hierarchical clustering of the text set, where the texts are described in the space of frequencies of the terms (the sequences) appearance.
4. Association of the cluster nodes to the terms, as semantic value of the node, thus, forming a thesaurus.

Textual representation also used by programmers using revision control systems to describe work done. The description can be considered as a text block of corresponding literate programming source code. There are developer groups, which have agreements of tagging text with special words (such as "UPD:", "TODO:", "FEATURE:") to define modification semantics more formally. Analysis of the descriptions allows one to connect ontological notions to source code components and functions of the new structures to its implementation.

The history of the development process is to be stored in a revision control system. Its branching structure will reflect the natural structure of the software development process. Comparing the branching structure with existing formal taxonomies gives rise of relation of the object classes to their implementation approaches. Open source distributed concurrent versioning system Git [9] has most powerful commit approach, which allows one to fix changes partially, and powerful branching model, merging, pushing/pulling changes, and repository cloning.

User interfaces are also the sources of the modifications as they are parts of the software reflecting all structures of the software projects. The main role of the user interfaces in the software development process is adaptation to the software structures and user requirements. So, the allowing user to modify the user

interface will result in the new set of modifications related to layout of the widgets, grouping the common components, and fine adjustment of the behavior of the widgets.

CONCLUSION

Software development life cycle has been considered as subject of the theory of complex systems and complexes [2] implying that the software development is a natural process. The life cycle is represented as system of models and morphisms between them. Analysis of the theory's properties realization in the model shown, that the present instrumental software productivity could be extended by means of developing techniques for analysis of the passed life cycle stages, analysis and propagation of modification of the models.

In the last section of the paper we considered some existing sources of modifications in the framework of Model Driven Architecture (MDA) and software utilization, for example, joining the code generation stage of MDA and compilation stage of programming language allow one to propagate modification of previously generated source code to the abstract models of the software; extraction formal taxonomy from analysis of textual representations of users' requirements and logs of concurrent versioning systems [9] allows one to figure out new notions from new requirements.

For some tasks appearing in the paper a variant of the solution is presented as a methods or informational technology. The problem of the software development history analysis is not considered and is a subject of further investigation, as well as implementation of the considered ideas as an open-source MDA software development tool.

REFERENCES

- [1] "Software development process - Wikipedia, the free encyclopedia", access date - 03-aug-2011, http://en.wikipedia.org/wiki/Software_development_process.
- [2] "Homology And Homoyopy in Geographic Systems", Scientific editors: A. K. Cherkashin, E.A. Istomina. Novosibirsk Academic Publishing House "GEO", Novosibirsk, Russia, 2009, 351 p. (in Russian)
- [3] D. A. Marca, C. L. McGowan. "SADT: structured analysis and design technique". McGraw-Hill Book Co., Inc.: New York, NY. 1988. 392 p.
- [4] D. S. Frankel. "Model Driven Architecture: Applying MDA to Enterprise Computing". Wiley Publishing, USA, 331 p.
- [5] "Formal methods - Wikipedia, the free encyclopedia", access date - 03-aug-2011, http://en.wikipedia.org/wiki/Formal_methods
- [6] E. A. Cherkashin, S. A. Ipatov. "Logical Approach to UML-model processing of Informational Systems" J. Conterporary Technologies. System Analysis. Modelling. 2009. N 3 (23). pp. 91-97. (in Russian)
- [7] K. R. Edward, "Leo's User Guide", access date - 03-aug-2011, http://webpages.charter.net/edreamleo/leo_toc.html
- [8] I.V. Zakharova, A.V. Melnikov, J.A. Vokhmitsev "An approach to automated ontology building in text analysis problems". Workshop on computer Science and Information Technologies CSIT'2006, Karlsruhe, Germany, 2006. P.177-178.
- [9] Jon Loeliger. "Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development". O'Reilly Media Inc., USA, 2009. 313 p.