Mathematica
Balkanica

New Series Vol. 25, 2011, Fasc. 1-2

Mathematical Models-Based Software Modernization [1]

Neli Maneva, Kraicho Kraichev and Krassimir Manev

Announced at the **Mathematics in Industry** *Conference, Sofia, Bulgaria, 11-14 July 2010*

Over the years of Software Engineering (SE) existence a great variety of models have been proposed for any type of SE objects – products, processes and resources. The models are different in volume, degree of formalism suggested as well as usefulness of the impart information. The current paper explains briefly a general models-based approach for managing SE activities and its application to a particular activity – software modernization. The software artifacts involved in a modernization activity and the models representing them are described. The results of an empirical study on the application of the proposed mathematical models in a real-life software modernization project are discussed.

AMS Subj. Classification: 68N30

Key Words: software engineering, mathematical models, software modernization, software evolution, software erosion

1. Introduction

Software engineering (SE) is a comparatively new research area, started in 1969 as an answer to the so called "software crisis". Unifying the efforts of people with both theoretical and software business background, this interdisciplinary field is a sophisticated mixture of scientific, technological and managerial methods and approaches so as to assure an efficient process for producing high quality software products. But according to some statistical data nowadays only 10% of the existing and already validated (formally or empirically) scientific methods are used n software development [7]. The SE scientists confess

[1]This work was partially supported by the Bulgarian National Science Research Fund through contract 02-102/2009.

that they often use too high level of formal methods description without clearly defined procedures whether and how the scientific results can be transferred to practice. The practitioner's point of view is that the software business is intensive, comprising too many and very complex projects, accomplished under strong constraints and insufficient resources – financial, human (both in quantity and qualification), time, etc. So any answer to the question how the science and business in the SE area should collaborate in order to be mutually enriched, will be highly appreciated by the SE community.

Searching for some science-based solutions, we developed a new approach, called 3M – Management through Models and Metrics. The general purpose of this approach is to try to assure continuously improved software development cycle for a stated real-life SE problem and identified goals to be achieved.

Next section comprises the essence of the proposed 3M approach and how it can be used for a specific SE activity, namely software modernization. In Section 3 the content of this activity is briefly described and the results of experimental software modernization for a real-life project have been presented. In the Conclusion some further research and development intentions are shared.

2. The 3M approach: Management through Models and Metrics

The idea for intellectually manageable software projects encouraged us to try to develop a new approach, meeting the modern management principles with the agile SE modeling [7]. We stated that the proposed approach should be:

- ***Scientific*** – to apply some already validated formal and rigorous methods. We believe in the power of innovations, based on solid theories and will try to overcome their restricted use till now, explained by scientific methods complexity, insufficient theoretical background of software practitioners and lack of automated tools, supporting such approaches;
- ***General*** – to be constructed in a way, assuring its feasibility for any SE activity, identified as significant;
- ***Flexible*** – to be efficiently tunable to different application contexts.

The detailed description of the 3M approach is beyond the scope of this paper, so we only mention its principles and the stepwise procedure for practical use.

The approach is based on the next three principles:

a) Interpretation – it follows the modern paradigm of scientific knowledge: to move from factologic description of the objective world to dialogic interpretations, made by a valuable subject with a specific point of view.

b) Reasonable choice – at each critical decision point to apply the scientific method of Comparative analysis [6] so as to assure systematic and efficient local optimization;

c) Measurement – involves the systematic use of software metrics, enabling software practitioners to gain insight into their work and products developed. All obtained measures can be analyzed further so as to provide assistance in management and technical activities.

From managerial point of view this means:

- at any moment modeling should take into account a preliminary defined ***goal*** and the ***available resources***;
- construction of ***multiple models***, presenting different aspects of the analyzed SE objects;
- ***model evaluation*** on its applicability, utility and validity – for continuous quality improvement [5];
- maintained repository of existing models and tools to create and use them.

We propose the following feasible procedure for the 3M-practical use:

Step 1. Analyze in-depth the SE activity under consideration. Identify the basic SE objects (products, processes and resources), which can be further studied. Clarify the amount of resources (people, money and time) needed.

Step 2. Repeat the following within the planned resources: state the goal, construct the model, apply the model, estimate the model through metrics;

Step 3. Summarize the knowledge and experience gained and describe the set of identified best practices for this SE activity.

Next we describe how the proposed 3M approach can be implemented for a specific SE activity – software modernization.

3. 3M approach to Software Modernization

3.1. Software system evolution

Software system evolution is the process of the initial development of a software system followed by its continuous change. These continuous changes can range from small maintenance fixes all the way to a complete reengineering of the entire software system. Thus the process of software system evolution

spans the entire time from the start of the development of the initial version of the particular software system to the end of its life.

Software system evolution activities can be divided into maintenance, modernization, and replacement [3]. Maintenance involves small and frequent updates of the system targeted at fixing bugs or introducing small pieces of new functionality. Maintenance does not mean major reengineering of the system. Modernization on the other hand consists of more massive changes compared to maintenance, but retains a significant piece of the existing system. Such changes could be improving the architecture of the software system or replacing and/or adapting a legacy software technology. Replacement is the reengineering of the system from scratch. It incurs a significant cost on the organization and involves a considerable amount of risk. That is why organizations are reluctant to re-implement the entire system from scratch.

Maintenance is an indispensable activity in the life cycle of a software system, but it leads to ***software erosion***. Software erosion is the constant decay of the internal structure of a software system. It takes place during every phase of the software development life cycle and is most evident as a result of software maintenance.

The major symptoms of software erosion include [12]:

- the system breaks in unexpected places after changes are made;
- the system is hard to change because every change forces many other changes;
- it's hard to disentangle the system into reusable components;
- system artifacts are hard to read and understand;
- the system grows increasingly isolated from the rèst of the organization's IT environment;
- adding new functionality becomes increasingly difficult.

As a result of the above, maintenance cost is rising and system development projects are usually running over time and over budget.

Next follows a brief description of some causes for software erosion [11, 12]. Knowledge about the internals of a software system decreases over time mainly because the involved designers and developers change jobs, have been retired and forget the details of implementation. Even the best documentation leaves out certain assumptions, bits and pieces, which tend to accumulate over time leading less-than-perfect understanding of the system. As a result the

initial design and best practices defined for the system tend to become less adhered to over time.

One of the goals of software modernization is to manage software erosion.

As it turns out modernization is the middle ground between maintenance and replacement as it allows organizations to achieve some of the benefits of replacement at a fraction of its cost. Modernization also allows a staged approach in which only part of the system can be improved at a time [2].

3.2. A stepwise 3M-procedure for software modernization

Traditionally software modernization has been confronted as an engineering and business problem as reported in [3, 9, 10]. Many mathematical models have been applied to SE activities in the past [7] but to the best of our knowledge mathematical models have not been applied in a consistent manner to software modernization in particular. The above-mentioned stepwise procedure for software modernization comprises the following steps:

Step 1 – the software modernization activity is studied and the goals, scope and goal-oriented artifacts are defined.

Goals

Every software modernization project is unique. Different organizations have different IT environments, different targets, different time-frames which result in very different approaches to software modernization. That is why it is very important to clarify and define the goals of the project at the earliest time possible. An example of a goal would be: "The mainframe-based IT system in plant X must be migrated to run on UNIX servers."

Scope

Based on these goals the scope of the project must be defined. The scope defines the set of sub-systems or modules which are subject to modernization. Some organizations choose a staged approach to modernization, in which the IT environment is modernized one system or sub-system at a time [2]. In a staged modernization the scope for each stage is different. The scope depends on the goals set and it is not unusual to go back and refine the goals already set because the scope they imply is infeasible, taking into account the available resources. Depending on the project and software system the scope can be defined in terms of sub-systems, modules, packages, libraries, IT environment specifics, etc. For example the scope of the above goal could be: "The procurement, accounting, manufacturing, warehousing and delivery sub-systems of the mainframe-based IT system must be migrated, but not the payroll sub-system."

Artifacts

Based on the goals and scope the artifacts for modernization must be defined. Examples of such artifacts are program source code, configuration data, structured and/or unstructured user data, documentation. Each of the artifacts must be described. Probably the most important part of this description is the target artifact, which must materialize as the project execution progresses. For example: "The legacy JCL-based batch processing must be migrated to UNIX BASH-based batch processing." Another important part of the artifact description is the size of the source artifact. For example: "The size of the source artifact is 1 million lines of code." Number of lines in the source code is just one of the metrics used. The metric is dependent on the type of the artifact. For example data in a relational database could be measured in number of tables, number of fields, number of rows, byte size of a database dump, etc. The size of the artifact is important, because it sets constraints on the transformation approach. For example, if the size of an artifact is significant it may not be feasible to employ a manual transformation and still meet the project goals for timeframe and budget. When the type of the artifact is data, then the artifact description must contain elements like source and target data structure, format, encoding, storage, backup and restore as well as disaster recovery policies, etc. For each artifact success indicators must also be defined.

Artifact classification

First we determine the available artifacts and classify them. We distinguish between two different types of artifacts – basic and derived. ***Basic artifacts*** are artifacts manually created by humans – for example, program source code, scripting code, configuration files, database data definition language scripts, manually written documentation. ***Derived artifacts*** are artifacts which have been created automatically from basic artifacts by software tools – for example, compiled program code, automatically generated source code, database dump files, automatically generated documentation.

There are two major differences between these two types of artifacts from the point of view of a modernization project. Basic artifacts are created by humans, which means that they contain much more information than do derived artifacts. Typical examples are program source code comments, variable and function names, manually written documentation, even source code which cannot be compiled with the latest version of the compiler. This feature makes them a good candidate for extracting knowledge about a software system. On the other hand precisely this feature of basic artifacts makes them difficult to

process with software tools. The reason is that basic artifacts can exhibit much less structure compared to derived artifacts, which is a feature of natural human languages and especially the English language. Typical examples are source code comments and manual documentation.

Derived artifacts are better structured than basic artifacts, because they are created in an automatic way. This makes them good candidates for machine (automatized) processing. On the other hand derived artifacts typically contain a fraction of the information available in the basic artifacts they started with. A typical example is source code comments, which are not available in the compiled executable code. Another example is that the structure of the database data can be missing from a database dump, whereas it existed in the original data definition scripts.

Thus both basic and derived artifacts provide pros and cons compared to each other. That is why the decision which artifacts to use for a particular part of the modernization project depends on what information the transformation needs in order to produce the target artifacts. If the information contained in derived artifacts is sufficient, then use them, because the data lends itself naturally to machine processing. If the information in derived artifacts is not sufficient, then basic artifacts or a combination of both basic and derived should be used.

Sometimes it happens so that the migration project doesn't have access to all the artifacts – for example some organizations don't employ configuration control and as a result some changes to the source code are lost and the only available up-to-date artifact is the executable program code. Another example is organizations which are unwilling to disclose their source code making the derived artifacts the only available to work with.

In a modernization project the artifacts are used over and over again to create and re-create the intermediate models used during the transformation process. Since normally the size of all artifacts as a whole is huge, the time necessary for software tools to build the models from the available artifacts can be significant and can sometimes be a bottleneck to the overall speed of accomplishing the modernization project. That is why speed is of the essence, which is another reason to use derived artifacts whenever the information they contain is sufficient for the desired transformation.

The complexity level of the performed analysis depends on the goals set as well as on the available artifacts. For example if the goal is to modernize a COBOL subsystem with the target artifact being Java code and the only available source artifacts are the executables of the COBOL programs, then the analysis will be more complex compared to the case where the COBOL source

code is available. The reason is that program, block and variable names as well as comments sometimes convey more information than the program source code itself.

The next activity is the classification of the available in-scope artifacts depending on their type into 3 groups – Programs, Data and Others. The Programs group can contain source code, scripting code, compiled executable programs, etc. The Data group can contain data definition scripts, database dumps, raw data, etc. The Others group can contain documentation, configuration files, log files, memory dumps, performance traces, user interface definitions, etc. The artifacts in these 3 groups are further classified into sub-groups according to different characteristics.

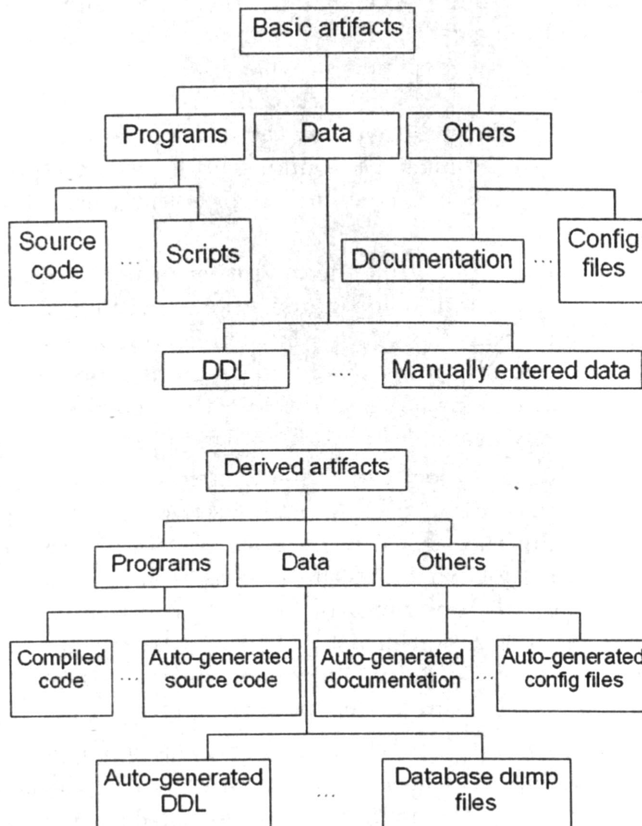

Figure 1: Artifacts classification

For the Programs group these characteristics can be programming language (e.g. COBOL, Assembly, C/C++, Java, etc.), execution environment (e.g. CICS, batch, .NET, App server, etc.) and so on. This classification depends on the specifics of the particular project.

For the Data group we classify the artifacts according to their structure – relational, hierarchical, object-oriented, unstructured, etc.

For the Others group we can classify the artifacts into the following subgroups: documentation, configuration files, log files, user interface definition files, etc. Memory dumps and performance traces can be grouped together with log files. Documentation artifacts can be further divided into internal (architecture, detailed design, admin guide, development guide) and external (user guide, tutorials, etc).

The reason for classifying the artifacts into different groups and subgroups is to enable the machine processing of the artifacts. The artifacts in a single group can be processed and analyzed by the same software tool with different parameters and/or modules used for the different sub-groups. This is so, because the operations the analysis tool applies on a single group or subgroup make sense for all elements of this group/sub-group. That is why this classification of artifacts defines a mapping between the set of artifacts and the set of analysis tools.

Step 2 – repeat the next activities within the available resources:

Modeling and Application of models

This step starts with building a 1-order (first order) model for each of the artifacts. For program code, this could be an AST (abstract syntax tree) based model [1], which stores the behavior of the program, but filters out syntax-specific information, which is irrelevant for the subsequent analysis. For data the model could be an entity model, which stores for example the name and attribute names and types of the entity (for example a relational database table), but filters out the machine-dependent representation, like file type, encoding, etc.

1-order models are built automatically by using software tools, which are configured appropriately. For example, the parsers for COBOL, Java and SQL are different modules of the same tool, even though they can still be generated by the same parser generator with the corresponding formal grammar. The classification of the artifacts described above allows to assign a tool and appropriate tool parameters to every artifact. Each of the modules of the tool can also receive different parameters. For example, to distinguish between different flavors of COBOL – ANS COBOL, COBOL 1985, COBOL 2002, etc. or SQL – SQL-86, SQL-92, SQL:1999, etc.

These 1-order models are independent of each other, they don't contain links/relationships between them. That is why they are considered at a lower level of abstraction, which is suitable for only basic analysis. So they are only an intermediate step on the way and not an aim in itself.

The next activity is to build higher-order models based on the 1-order models. New models represent a higher level of abstraction [4]. For example, control flow graph models are 2-order models which provide information on the pieces of the program, which can be executed runtime. These models can be used to pinpoint pieces of a program which can never be executed – so called dead code. An example of a 3-order model is a trace flow graph model, which captures additional information related to the actual execution of a program. This additional information could be time stamps (real-time or abstract clock cycles), memory-access data, etc. As shown in [8] this information can be used, for example, to make conclusions about the performance of the program, or to assess its currency. These 3-order models differ from the 1-order (and some 2-order) models – they incorporate information about the relationships between the different 1-order models – global variable references, inter-program dependencies, database access by the programs, etc. This additional information makes it possible to gain understanding about the software system which cannot be done with the models of lower orders.

These higher order models are then used to create a visual representation of the software system. Software architects and systems analysts can use this visual model in order to understand the software system. This is the part where human involvement is crucial and ultimately determines the results of the analysis phase. Software people determine the direction subsequent iterations of the analysis will take – set of artifacts to be analyzed in more detail, set of artifacts to exclude, what kind of analysis to perform, etc.

After the architects and analysts are satisfied with the level of understanding of the system they have gained, the transformation of the system begin. This means that the target artifacts must be configured. For example, such configuration could mean that for this functionality/logic Java code for the JBoss application server and SQL code (both DDL and DML) for the MySQL database must be generated. After this configuration is done, the software tools are run to make the generation. Usually at least several iterations are needed in order to fine-tune the parameters so that the result is optimal regarding the software architect's requirements.

Next a model is built, which represents a higher level of abstraction. Building of the model is automated by custom-developed software tools.

What follows is the evaluation of the software system based on the information in the model and the assessment of the degree of software erosion for each component in particular. The result of this step is a list of components sorted by the degree of software erosion. Based on this list a few components are selected in order to be analyzed further. After the analysis the components are worked on in order to be improved. These improvements range from the relatively simple ones like dead code elimination to more complex like refactoring of an entire sub-system to the extreme. For example, in some cases rewriting from scratch is the only viable option.

After these improvements the component is evaluated again in order to get valuable feedback what the effect was and to plan the next iteration accordingly.

We need different models for two main reasons. The first reason is that a typical enterprize software system comprises artifacts of various complexities. The second is that we need models at different levels of abstraction. While some concepts can be analyzed at a lower level of abstraction, others cannot be grasped until the presentation is on a sufficiently high level of abstraction.

Result interpretation, estimation of models, performance and achievements evaluation through a set of metrics After the models have been applied the result is measured using predefined metrics. The values obtained (measures) need to be interpreted as to how "good" they are, how "close" to the specified goal they are. The goal is usually defined as a set of metric values, which are aimed for. Thus the result interpretation is usually defined as a function which measures "distance" from the current metric values to the desired metric values. Based on the value of this function for the current metric values, the configuration parameters of the models are adjusted and the procedure is restarted.

Step 3 – incremental construction of the set of best practices for software modernization

As a result of the repeatable execution of the activities in step 2 described above, a set of best practices for this modernization of the particular artifacts gradually emerges. This "trial and error" approach results in accumulation of knowledge and best practices which can lead to more efficient execution of a similar modernization project in the future.

4. A Case Study: The 3M software modernization of a real-life project

The IT department of a multinational manufacturing company has been developing internally and maintaining the company-wide software system for over 30 years. This system handles every business activity in the company – raw material procurement, product development and testing, manufacturing, quality control, warehouse management, goods deliveries, customer management, accounting, etc.

The system was based on mainframe technologies – the z/OS operating system, DB2 for z/OS, CICS Transaction Server, CICS and Batch COBOL (8 million lines of source code), BMS, JCL (0.5 million lines of source code), Application System code (0.5 million lines of source code), REXX and Assembler for mainframe.

The difficulties associated with these legacy technologies were the significant maintenance costs related to hardware and software license fees, the decline in the number and availability of skilled mainframe technology experts to maintain it as well as the complexity of enhancing the mainframe applications in order to meet the evolving needs of the company. As a result of these obstacles the IT department of the company decided to undertake a complete migration of their software system to a modern hardware and software platform.

These IT assets were migrated to IBM System, running the AIX operating system, DB2 for AIX, TXSeries for AIX, COBOL for AIX, Bash shell script, Information Builders WebFOCUS, Java code and C code. A set of custom tools were developed and used for automating the translation of COBOL for mainframe code to COBOL for AIX code as well as the translation of Application System code to WebFOCUS code. This automated approach allowed the translation of the huge source code base to be completed much more quickly and efficiently than would have otherwise been possible.

The mainframe applications have been developed and maintained for over 30 years by different people. The growth of the system over the years had led to insufficient documentation. As part of the migration project the documentation was improved.

As a result of this migration the manufacturing company has achieved a significant decrease in IT operating and maintenance costs and a much more flexible and future-proof IT environment, much better suited to address the dynamic requirements of the enterprise.

The modernization project results are quite encouraging because they confirmed the feasibility of our 3M approach to a specific SE activity, namely software modernization. It is possible, although very unlikely, that the particular

industry project chosen is not representative of industry projects in general due to business processes and/or technological base being too specific, but anyway, we obtain at least one positive feedback from the practice.

5. Conclusion

The described 3M-method unifies modeling, measurement and management in the field of software engineering. This method seems to be a scientific, general and flexible approach to any SE activity under consideration, supporting its automated and efficient performance. We examine the approach in one modern and crucial for the software industry activity – software modernization. It has been chosen, because our investigation shows that many mathematical models have been developed to be used in different SE activities [7] but to the best of our knowledge such models have not been applied in a systematic and consistent manner to software modernization in particular. An empirical study on a real-life software modernization project has been conducted too, in order to validate the proposed mathematical models-based approach and to examine it in an industrial setting.

Some possible ideas for further research and development can be:

- to improve the modeling process and the procedure for its application, expanding the third step – the description of identified best practices;
- to increase the level of automation through a set of software tools, integrated in a framework;
- to continue the experiments in software modernization for projects in a few other application areas so as to analyze the results from the point of view of different stakeholders: users, developers, managers, etc.

References

[1] D. Binkley. *Source Code Analysis: A Road Map. Future of Software Engineering (FOSE'07)*, IEEE-CS Press, 2007.

[2] M. Brodie et al. *Migrating Legacy Systems: Gateways, Interfaces & Incremental Approach*, Morgan Kaufmann publishers Inc., San Francisco, California, 1996.

[3] S. Comella-Dorda, K. Wallnau, R. Seacord, J. Robert. *A Survey of Legacy System Modernization Approaches*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 2000.

[4] R. Kirkov, G. Agre. Source Code Analysis – An Overview, *Proc. of Cybernetics and Information Technologies 2010 (CIT 2010)*, **10**, 2010, 60-77.

[5] N. Maneva. Software Engineering Models and their Evaluation, *Mathematika Balkanika, New Series*, **18**, 2004, 149-156.

[6] N. Maneva. Comparative Analysis: A Feasible Software Engineering Method, Serdica Journal of Computing, **1**, No. 1, 2007, 1-12.

[7] R. Pressman. *Software Engineering: A Practitioner's Approach*, McGraw-Hill, New York, 2005.

[8] A. Rountev, K. van Valkenburgh, D. Yan, P. Sadayappan. Understanding Parallelism-Inhibiting Dependences in Sequential Java Programs, *Proc. of the 26-th IEEE International Conference on Software Maintenance*, Timisoara, 2010.

[9] R. Seacord et al. *Modernizing Legacy Systems*, Addison-Wesley Professional, Boston, 2003.

[10] N. Weiderman, V. Bergy, D. Smith, S. Tilley. *Approaches to Legacy System Evolution*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1997.

[11] `www.hello2morrow.com`, last visited October 2010.

[12] A. von Zitzewitz. *Golden Rules to Improve Your Architecture*, published on `www.hello2morrow.com`, 2008.

Institute of Mathematics and Informatics – BAS,
Sofia 1113, Acad. G. Bonhev str. bl.8, Bulgaria
e-mail: nely.maneva@gmail.com

Received 10.09.2010

Musala Soft Ltd.,
Sofia 1057, 36, Dragan Tsankov blvd., Bulgaria
e-mail: kraicho.kraichev@musala.com

Faculty of Mathematics and Informatics, Sofia University
Sofia 1164, 5, James Bourchier blvd., Bulgaria
e-mail: manev@fmi.uni-sofia.bg