МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2012
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2012
*Proceedings of the Forty First Spring Conference
of the Union of Bulgarian Mathematicians
Borovetz, April 9–12, 2012*

AUTOMATED SOFTWARE REENGINEERING MODEL AND FRAMEWORK[*]

Todor Cholakov, Dimiter Birov

This article represents a complete model for automated reengineering of legacy software systems. It describes in details the processes of software translation and refactoring and the degree of automation that these processes may achieve. In regards to the software translation process it introduces a reengineering pattern concerning pointers and address arithmetic. It also defines a complete workflow for the reengineering process and the possibilities for further development of tools concepts and algorithms.

1. Introduction. In this article we present a complete environment for automated software reengineering.

Software reengineering is the process of renewing an existing piece of software by migrating it to a new hardware or software platform, applying newer technologies or applying large transformations to the design and implementation to the software. The purpose of the software reengineering process is to renew a legacy system, improve its quality or features, migrate it to more modern hardware or software technologies and to improve its maintainability. There are several steps in this process and we are going to present a model for an environment and solution that would ease and automate each of them. These steps are *source code translation, restructuring, refactoring and changes in documentation.* After completing them, we will have a new working software system.

2. Source code translation. Legacy systems are often written in languages that are rarely used at present. Supporting such a system costs a lot because the number of IT professionals that have the knowledge to maintain it is small and decreasing. And more – the older languages may lack some features (such as objects or garbage collection) which otherwise would greatly improve the maintainability of the system. The decision in these situations is to translate the system to a newer programming language.

Usually it is not possible to automate the transformation process completely, because there may be concepts in the source language that are not present in the target one (for example pointers and address arithmetic in C++ is not present in Java). However, it is possible to automate most of the process and leave the developer to cope with the details.

[*]**ACM Classification:** D.2.7.
Key words: restructuring, reverse engineering, and reengineering.
The authors gratefully acknowledge the financial support by the Bulgarian National Science Fund within project DO 02-102/23.04.2009.

The structure of such a tool greatly resembles the structure of a compiler. The first step is to parse the source code and produce an abstract presentation of the code – a parse tree. This can be easily performed by using some common tools [1]. If the source and target languages are quite similar, the tool may stop to this step and then use the parse tree and transformation rules to produce a source code in the target language. This technique is applicable for highly similar languages like C and Pascal or when the target language extends the original one (for example Pascal code may be easily transferred to C++ code, because C++ has all the features of Pascal and has some additional extensions).

For those languages that the source cannot be transformed in this way, the tool must go one level deeper and produce a code in an intermediate low level language. The purpose of this step is to eliminate some of the more complex concepts like objects, classes and polymorphism. At this point the code works with simple data structures and method calls. Then, the intermediate code may be translated back to the target language by using simple rules for grouping the intermediate instructions into target language ones. This method can handle all language constructs except address arithmetic.

Translating address arithmetic is not possible in general, because a pointer can point to any place in memory. However, in some cases it is possible to translate address arithmetic operations if we make sure that the pointer always references some kind of array. In this case the address arithmetic is translated to array indexing.

A more complex case is when the pointer references an internal structure of an object or record. Sometimes it is possible to translate the result into object and a reference to one of its fields. If that is not the case, a serialization mechanism may be needed to transfer the original object into a byte array corresponding to the original structure of the object. Then, the address arithmetic operations may be completed and if there is a change to the object's data, the data in the array must be extracted back to the original object. For example consider the following code snippet:

Original code	**Target language code**
class DataObject{	*class DataObject*{
int a;	*int a;*
byte b;	*byte b;*
double c;	*double c;*
long d;	*long d;*
}	}
.....	.....
DataObject original;	*DataObject original;*
void pointer = &original;*	*byte[] originalPnt = serialize(original);*
pointer = pointer + 5;	
**pointer = 176;*	*int pointer = 0;*
....	*pointer = pointer + 5;*
	originalPnt[pointer] = 176;
	extract(originalPnt,original);

Fig. 1. Pattern for source code transformation when having address arithmetic

The serialize and extract functions may become quite complex and must implement all the knowledge about how the fields are presented and located in the original language. As a whole this method is quite inefficient, but may appear to be the only possible in some cases.

As the resulting code is often inefficient all possible optimizations must be applied on the intermediate code before translating back to the target language.

Additional requirement for the translation tool is to preserve all comments in the original code to their appropriate places in the target code, as well as to support linking the target language statements to the original ones. This would allow for the developer to track what happened through the code and optimize or tune up additionally some parts of the code, having greater meta knowledge about the system.

3. Restructuring. After the software is translated to the new platform, it may have a room for restructuring.

Restructuring is the process of reallocating software pieces into files and folders in a way that is more suitable for the new programming language or platform that is used. For example in C and C++ there is no concept like *packages* in Java and all the sources may be in a single folder. Restructuring also may divide a file into smaller files or even unite files if needed. Some of these activities may also be performed during the refactoring step and the used technology is the same as the one used in refactoring. The main difference to refactoring is the purpose. While the purpose of the refactoring process is to improve the existing design of the code, the purpose of the restructuring is to create as good initial design as possible according to the new platform. For example, software written in procedural language may end as a single Java class after translation. From this starting point the software may be restructured into several classes. A good algorithm for doing this is to explore the interconnections between the different methods and the data they use [2].

For example, if there are several methods that call each other using the same set of parameters, then this may indicate that they should be extracted as a separate class, and the data itself to be made as fields in the class. Data pieces that often go together may indicate that they must belong as fields of a single more complex structure.

As a result of restructuring the code, we get a more readable code corresponding better to the technologies of the target language.

4. Refactoring. The next step in the reengineering process is the refactoring. The refactoring is the process of improving the readability, maintainability or performance of the code, without changing its behavior.

There are three aspects when talking about refactoring:

- how to define the term “behavior of a code”;
- how to recognize the need for refactoring for specific piece of code;
- how to perform the refactoring itself.

The most common response of the first question is to consider the piece of code as a black box [3]. We consider that a transformation preserves the behavior, if for each set of input data the original and the transformed software give the same results. Though this definition is very simple, it is quite difficult to prove for more transformations if they preserve code behavior or not. For example, when the same expression is calculated twice, we may consider assigning it to a variable. But what happens if one of the used functions has a side effect? Although the expression itself has the same value, at a later place the behavior of our software may change. Such situations are difficult to recognize, so even the simplest refactorings do not completely guarantee preserving the source code behavior.

Other aspects of the software behavior that are not covered by the previous definition are the speed and security of the system.

The need for refactoring of the system may be determined in two ways.

The first one is by applying some software quality metrics. These metrics may perform a static analysis of the code and discover parts of the code which need improvement of some kind. Another type of metrics are the dynamic ones. These analyze the code behavior while the system is running. Usually these metrics discover performance issues.

The second way for determining refactoring needs is the bad smells. Martin Fowler [4] defines more than twenty symptoms that some piece of code needs refactoring. Although most of these symptoms are not formally defined, they may be implemented with different degree of complexity as metrics.

The last question about refactorings is how to perform the refactoring itself.

Each refactoring may be considered as consisting of three parts – precondition, algorithm for applying the refactoring and post-condition.

The precondition defines under which circumstances the refactoring may be applied. For example, in order to apply the "extract method" refactoring the precondition may be that there is no more than one local variable that is changed in the block to be extracted and used outside that block. Also a method having the same signature as the extracted one must not exist.

The algorithm defines the refactoring as a sequence of simple code transformations.

The post-conditions check the validity of the refactoring. Usually post-conditions are not defined. They are needed only for refactorings where it is easier to perform the refactoring and check if it is valid, than to check the precondition.

There are several degrees of automation of the refactoring process [3].

The highest degree is the fully automated refactoring systems. These systems discover the needs for refactoring of the software and apply the needed refactorings automatically. They implement a set of metrics and algorithms for recognizing bad code pieces and after such a piece is discovered, the system applies the refactoring automatically.

The main disadvantage of such systems is that they must be very conservative when determining whether to refactor a piece of code or not in order to preserve the existing behavior of the code. Such systems usually perform a very small subset of refactorings.

The main advantage of these tools is that they may save a lot of time which would be otherwise needed to discover and apply the refactorings especially in large software systems.

Example for such system is the GURU tool for refactoring programs written in Self [5].

The second degree of automation concerning refactorings is the interactive systems. These systems analyze the source code and give suggestions about the refactorings that may be performed. Then the developer chooses some of the refactorings and the system applies them on the code. These systems may suggest a much wider set of refactorings and even suggest refactorings that may change the original behavior of the code. They are able to save a lot of time for recognizing refactoring needs and performing the refactorings but still need a human interaction to decide which of the suggestions are to be applied.

The simplest degree of automated refactoring systems is the semi-automated systems. These systems do not implement algorithms for discovering refactoring needs. The developer is responsible for discovering the piece of code that needs refactoring and the suitable refactoring and then the system performs the refactoring automatically. These systems usually are implemented as modules of development environments such as Eclipse [6] and IntelliJ Idea. They are very suitable for day-to-day refactorings but cannot be used for refactoring a large software system.

5. Data reengineering. Changing the language of the software and refactoring it may improve its maintainability, but often there is also need to change the presentation of the data used. This task has its own challenges and usually may not be fully automated. However, there are tools that can ease the process.

The most difficult case concerning data migration is when the data of the legacy system is stored in a set of files that have specific format or structure. Usually it requires a lot of effort to migrate this kind of data to relational or object oriented databases. Tools like Xenomorph Timescape [8] ease the process by taking the migration effort out of the source code, thus allowing the developer to migrate the existing data to a new format or persistent database but leaving to the end user the ability to provide data in the old format.

Usually there are common tools for migrating data between most widespread database engines, so if the legacy data is stored in a database the task for migrating the data itself is much simpler.

The part of the data migration process that cannot be automated is the data usage in the source code itself. Even the simplest solutions (like implementing a data abstraction layer that translates old data manipulation routines into calls to the new data layer) require a lot of manual development

6. Documentation. End user documentation should not need to change in the reengineering process. But that is not the case when talking about the technical documentation. Usually the technical documentation describes the parts of the system, the technologies used, the data required and produced and the connections between the modules. In the reengineering process this documentation will have a lot of changes. Only very small part of them may be automated (relocating module descriptions and connections between them may be automated when the restructuring and translation tools do their work, if suitable connections between the code and the documentation are defined beforehand).

7. The complete model. We defined the activities in the reengineering process and now we are going to represent a complete model of a system for automated (at the best possible degree) software reengineering. The concrete set of tools for such a system will differ according the specific needs and constraints of the reengineering project, but the general workflow and requirements remain the same:

- A tool for language translation. This tool takes the legacy system as an input and produces as output a system having the same behavior, but written in the target programming language. The structure of the system is changed as little as possible. All comments in the code should go to the target system. Connections between the code and technical documentation (if any) must also be preserved.
- Development environment for the target language. At the end of the translation

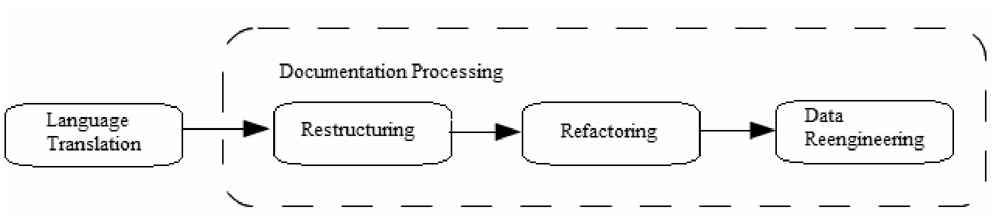

Fig. 2. Reengineering workflow

step the developer may need to additionally tune up the resulting code. Depending on the efficiency of the translation tool this may take a lot of effort.

- A tool for automatic restructuring of the target system. This tool must analyze the source code and redistribute it in modules or classes, making it more suitable for the target technology used. The tool must also maintain the technical documentation keeping it in sync with the changes in the source code.
- The developer may need to manually do some of the restructuring not performed by the automated tool. A semi automated refactoring system for the target language (integrated in the IDE if possible) is needed for the task.
- An automated refactoring system to perform as many refactorings for the code, as possible.
- An interactive refactoring system, which is able to suggest a wide set of suitable refactorings.
- A data migration tool to migrate the data to its new format or storage. The changes in data formats must be reflected in documentation.

As a result of the above workflow a system working on a new software and data platform is produced.

8. Further improvement. There is a lot of room for improvement and development in the specified model in several directions:

- A more efficient algorithms and techniques for language translation are needed in order to reduce the additional time needed to tune the resulting code.
- New algorithms and metrics for automated refactoring tools may increase their applicability.
- Improvement of the algorithms for applying the refactorings themselves. This includes for example an algorithm for determining a suitable name for an extracted method.
- A system for generally tracking changes in source code into documentation and its connection to translating and restructuring tools is needed.

9. Conclusion. We defined the process of reengineering of legacy systems and described in some details the processes of refactoring and source code translation. When analyzing the whole workflow the overall conclusion is that a large part of the process can be automated to some extent. The further goal is to improve the described tools (and maybe introduce some new ones) in order to reduce the development time needed for reengineering a legacy system.

REFERENCES

[1] A. Aho, M. Lam, R. Sthi, J. Ullman. Compilers: Principles, Techniques, and Tools. Pearson Education, Inc, 2006

[2] I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In: *Proc. of the 11th Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, 1996, 235–250

[3] T. Mens, T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, **30** (2004) No 2, 126–139.

[4] M. Fowler. Refactoring – Improving the Design of Existing Code. Booch Jacobson Rumbaugh, 1999.

[5] I. Moore. Guru – A Tool for Automatic Restructuring of Self Inheritance Hierarchies. PhD Thesis, University of Manchester, 1995.

[6] T. Widmer. Unleashing the power of refactoring. Eclipse Magazine, July 4, 2006.

[7] `http://www.jetbrains.com/idea/webhelp/refactoring-source-code.html`

[8] Timescape Data Unification – `http://www.xenomorph.com/downloads/whitepapers/timescape-data-unification/`

Todor Cholakov
Dimiter Birov
Faculty of Mathematics and Informatics
St. Kliment Ohridski University of Sofia
5, James Bourchier Blvd.
1164 Sofia, Bulgaria
e-mail: todortk@abv.bg
e-mail: birov@fmi.uni-sofia.bg

МОДЕЛ И СРЕДА ЗА АВТОМАТИЗИРАНЕ НА ПРОЦЕСА НА РЕИНЖЕНЕРИНГ

Тодор П. Чолаков, Димитър Й. Биров

Тази статия представя цялостен модел за автоматизиран реинженеринг на наследени системи. Тя описва в детайли процесите на превод на софтуера и на рефакторинг и степента, до която могат да се автоматизират тези процеси. По отношение на превода на код се представя модел за автоматизирано превеждане на код, съдържащ указатели и работа с адресна аритметика. Също така се дефинира рамка за процеса на реинженеринг и се набелязват възможности за по-нататъшно развитие на концепции, инструменти и алгоритми.