УДК 004.416.3+004.4’242

Automated Semantics-Driven Source Code Migration: a Pilot Prototype

Artyom Aleksyuk (Peter the Great St. Petersburg Polytechnic University, Russia; JetBrains Research)

Vladimir Itsykson (Peter the Great St. Petersburg Polytechnic University, Russia; JetBrains Research)

The purpose of the study is to demonstrate the feasibility of automated code migration to a new set of programming libraries. Code migration is a common task in modern software projects. For example, it may arise when a project should be ported to a new library or to a new platform. The developed method and tool are based on the previously created by the authors formalism for describing libraries semantics. The formalism specifies a library behavior using a system of extended finite state machines (EFSM). The mentioned EFSMs are a foundation of the code migration method.

This paper outlines the metamodel designed to specify library descriptions and proposes easy to use domain-specific language (DSL), which can be used to define models for particular libraries. The mentioned metamodel directly forms the code migration method which is also described in the paper. A process of migration splits into five steps, and for each step the algorithm was developed.

Models and algorithms were implemented in the prototype of an automated code migration tool. The prototype was tested on both artificial code examples and several real-world open source projects. Results of the experiment indicate that code migration can be successfully automated with developed tool acting as the proof of concept. Models and methods designed form a basis for more powerful migration methods and full-featured automated code migration tools.

Keywords: software library, code migration, behavioral description, program transformation

1. Introduction

This study is devoted to automated program migration, which is an actual software engineering task. Usually developers are faced with this task when they are going to upgrade their project to a new version of library or port their projects to a more effective, more secure or more functional library. Usually the source project has been properly tested and the developer wants to be sure that the ported project has an equal level of quality. A manual migration does not guarantee the expected quality level. Developers have to test the migrated project from scratch as a new one.

The aims of our research are

- to develop the new migration method based on formal library specifications which is fully automated and preserves the quality of the project;
- to prove the feasibility of our approach.

The proposed approach is based on previously created by the authors formalism for library specification description. The mentioned formalism represents libraries as a set of extended finite state machines (EFSM). Each EFSM reflects the lifecycle of the whole library or its entities. Examples of such entities are statically or dynamically created files, sockets, semaphores, streams, mutexes, threads, etc. States of EFSM are states of objects, vertices represent library calls. Full specification of the formalism can be found in [1].

One of the main aims of this work is to develop an experimental tool which can be used for proving feasibility of automated model-based migration. To do this we designed a simple DSL for specifying libraries' behavior. These specifications are used as an input for the multistep method which transforms source project into the new one according to models of both the old and the new libraries.

To evaluate our approach, we have conducted a set of experiments with artificial and real-world open source projects. All examined projects were ported correctly by our tool. These results are a proof of applicability of the proposed approach.

The rest of the paper organized as follows. The first section contains the description of the state of the art. In the second section the proposed approach to migration is introduced. The third section is devoted to the implementation of the tool prototype that proves feasibility of our approach. In the fourth section developed tool is evaluated on the artificial and real-world projects. In the conclusion the obtained results are analysed and possible directions of a future research are discussed.

2. State of the Art

Firstly, we need to outline the migration task. Suppose we have a software system using a specific set of libraries. The task is to port a program to another set of libraries without rewriting the system from the scratch. Below we consider several approaches to an automated solution of this task:

- migration with a help of manually written wrapper libraries;
- migration by running a syntax-level porting tool;
- migration by applying a semantics-level porting tool.

One of the most common ways to migrate a program to another library is to use manually written wrappers [2]. Wrapper is a small library which has the same interface as the source library and uses the destination library to actually perform a work.

Complexity of wrappers may vary a lot. The simplest ones just redirect function calls. This approach is sufficient if a destination library has functions with the same set of arguments and has the same semantics. More complex wrappers may also apply data format transformations.

There are several downsides of the wrapper approach:

- wrappers are usually written for specific source and destination libraries;
- complex wrappers can severely affect the performance;
- wrappers usually restrict access to destination library's structures and/or methods;
- if some parts of the program directly use a destination library and other parts use it through a wrapper, it makes harder to exchange data structures between these parts.

Despite all the limitations listed above the wrapper approach can be utilized for some libraries. Several known wrapper projects are listed below:

- ANGLE is an OpenGL ES implementation on top of the Direct3D [3];
- SLF4J (Simple Logging Facade for Java) is a unified logging framework which can use several logging implementations as a backend [4].

Migration by running a syntax-level porting tool is another available approach. It differs from the wrapper approach in that this one actually ports software by changing its source code. These tools primarily rely on the information about a syntax of programming language and are based on template replacement or term rewriting.

One example of usage of the template replacement method is IntelliJ IDEA [5] with its built-in feature called Structural Search and Replace. This feature is very similar to a familiar search and replace tool found in most text editors, however it is aware of a programming language syntax. For example, it is able to find code which has another formatting style. Also, it can match code with small insignificant differences such as class fields declaration order.

An example of a term rewriting software is TXL tool [6]. It relies on its own functional language for specifying rewriting rules. To define a programming language syntax it uses the extended Backus–Naur form. The development of a term rewriting approach leads to appearance of rewriting strategy concept implemented in Stratego/XT [7] and DMS [8] tools.

An example of a practical usage of this approach to migrate program is described in [9]. The authors of this paper wrote a set of rewriting rules to help migrate programs from Qt 3 to Qt 4. During the evaluation phase, they managed to partially migrate kdelibs project to a newer version of Qt library.

The main drawback of the syntax-level migration approach is an ability to perform only the simplest replacements like method name change or call arguments reorder [10]. More complex replacements require to write large templates or are totally impossible with this approach.

The approaches described above have a limited power to automatically migrate source code in case of a significant difference between a source and a destination libraries. An additional information such as a library semantics is needed. In this paper, we use a more powerful and universal semantics-based approach.

One of the recent articles [11] extends the pattern-based approach with semantics information to better match program elements which needs to be migrated. However, the replacement process is still controlled by the set of templates and so is not capable to perform complex changes to the source code.

3. The Proposed Approach

The key part of the proposed migration approach is a library metamodel. The design of the metamodel is mostly influenced by the following criteria:

- the complexity of the analysis;
- the ability to express libraries' semantics;
- the ease of library model construction.

In this paper, we use the previously created by the authors formalism [1] as a base for the metamodel. The mentioned formalism was created to be used in a wide range of areas, including the defect detection and software mining[1].

The metamodel is a set of EFSMs (extended finite state machines). Each EFSM represents a semantic entity which library can process or use for its operations. For example, a File I/O library will possibly have entities named like "File", "Filename", "Stream", etc. Usually each class in a library has a corresponding EFSM, but this is not a strict rule. EFSMs are defined

[1]It should be noted that the formalism is still being developed and should not be treated as a final work

by tuple $< Q, Q_0, X, V, C, C^A, C^D, U, F, T >$.

Each library's EFSM has a set of states Q. For example, the "File" automaton can have states like "Opened" and "Closed". Also, each automaton declares a set of transition relations T and the non-empty set of initial states Q_0. X is the set of finish states. An example is a *Closed* state for a *File* entity. C is the set of function calls, constructor calls and other code elements acting as stimuli for state transitions.

Some code elements return new entities after execution. C_i^D is the set of child automatons created when C_i code element is being activated.

Most complex libraries cannot be fully described with just states and transitions, so we decided to extend the library metamodel with properties V and semantic actions C^A. Each EFSM may have a set of properties identified by name and containing an arbitrary value (strings, integers, etc.). Properties may be modified by update functions U. Transition between states is only possible if guard condition predicate $F(V)$ is true.

Metamodel actions define semantically significant events which cannot be expressed by a property change or a state transition. Actions are implemented as a transition attribute. Also, they may have arbitrary parameters just like properties. For example, a change of library settings may be defined by an action. More details about actions and properties may be found in the article [1].

4. Method

The proposed migration method consists of five steps.

The first step is the **trace extraction**. A trace contains a list of code elements placed in the order of activation. Any suitable approach can be used to do this task.

The second step is the **trace mapping**. This step includes fetching the program trace and mapping it onto the source library model.

The third step named **equivalent trace calculation** is the most interesting one. Equivalent trace calculation is an immediate process of searching a replacement for a source model trace. The resulting model trace should use a destination library and must be semantically equivalent. One of the most important advantages of the proposed method is that a transformation is done in the metamodel context.

The replacement search is driven by two factors. Firstly, if a source trace contains entity creation, a resulting trace must also have it. For example, if a source (migrated) program fetches a file length, i.e. creates a FileLength entity, a resulting trace should also create this entity. This is necessary because a source program may pass the created entity to another function or store in a class field, and migrated program must do the same. Secondly, the resulting trace must contain the same set of actions as the source one.

The algorithm currently used by the tool is based on a breadth-first search (BFS). The library model is treated as a graph and solution of the shortest path problem can be viewed as a replacement sequence, where the sought-for vertex is a required state of the EFSM. The search is simultaneously started from several vertices (all library entities available in the scope). If the equivalent path is not found, a user help is needed.

At first, the library model is being converted into the graph representation. Each extended finite state machine is transformed into the separate graph, and then all graphs are merged into the single one. The resulting graph contains several vertex clusters which correspond to each EFSM.

Actions and properties from the library model put additional restrictions for the acceptable solution. To take these restrictions into account, we define a new graph G' where each vertex of G' is a possible state of a traversal (after visiting some number of edges), and where there is an edge $u \to v$ if starting from state u it is possible to add one edge to the path to reach state v.

Traditional BFS algorithm keeps two queues: a visited queue which contains vertices which were already visited and a pending queue which contains vertices going to be processed. Our algorithm has a third queue named "have missing requirements" which contains models with inaccessible dependencies. If the visited queue receives a model which has a needed dependency in it's context, these two models are merged and placed in the pending queue.

The pending queue is sorted so the models with a shortest path have a highest priority. As metamodel edges are unweighted, the found path is minimal.

As we mentioned above, the next edge in the path does not necessarily starts from the end vertex of the previous edge. In this case a following step is applied. For each vertex in the context an additional model is generated which has a current vertex set to a fetched one. All generated models are placed in the pending queue. This step allows to fallback to any entity available in the context. In some cases, a number of generated models may be huge so this step is applied only in exceptional cases, for example, if there is no other solution available.

Summing up, results of this step are:

- the abstract syntax tree (AST) node which should be replaced;
- the edge from the source library matching the replaced node;
- the sequence of edges from the destination library's model which forms the replacement.

The fourth step is the **mapping of a new trace back into AST nodes**. This task can be easily done using information from the library model.

The fifth step is the **program transformation**. A set of AST nodes received from the previous step are placed in the syntax tree.

5. Tool Development

To demonstrate the feasibility of automated code migration and test the metamodel, the proof-of-concept tool for program migration was developed[2].

In this work, we have decided to use Java programming language for analyzing and processing. The main reason is an absence of "hard to analyze" constructions like macros, class templates, operator overloading, etc. Processing of these construction does not make sense for the proof-of-concept tool.

We have reviewed several approaches to fetch a program trace:

- using JVM application programming interfaces (APIs) to instrument program execution;
- interacting with existing Java debuggers;
- using aspect-oriented programming (AOP) to weave instrumentation code in the program.

The developed tool uses aspect-oriented programming because there are several widespread and well-tested AOP implementations which allow to instrument code with minimal efforts [12]. We have chosen *AspectJ* implementation as the most popular one.

We use *JavaParser* library to parse Java source code. It provides a high-performance and easy to use tool which is able to transform Java code into the AST model. It can also transform AST back into a code, preserving comments and formatting, which is very helpful for transformation tasks. Preserving formatting is necessary if the source is managed by a version control system like Git.

The developed tool relies on a custom domain-specific language (DSL) for library model description. The DSL is based on *Kotlin* programming language [13] and allows to write models without a deep knowledge of tool architecture.

[2]Source code of the proof-of-concept migration tool is available at https://github.com/h31/LibraryMigration

The developed tool contains the module which allows to visualize library models. It helps a lot to debug models and to communicate with persons who have an expertise in a library but does not know much about DSL.

6. Evaluation

First, it was necessary to choose a set of libraries all of which solve the same problem. Our choice was HTTP client implementations. We made models for the following libraries:

- *Apache HttpClient* from *HttpComponents* project;
- Java Class Library built-in client (*java.net.HttpURLConnection* and related classes);
- *OkHttp* client.

All of these libraries have a set of common entities (URL address, HTTP header, response content) and a set of library-specific entities. For example, Java built-in client has the class named *HttpURLConnection* which combines both the request and the response. *Apache HttpClient* and *OkHttp* contain the dedicated request and response classes, through slightly different. Also, these libraries have the Client object which should be instanced to make an HTTP request. Java built-in client does not have such a class.

After the first pilot version of tool was finished, we have tested it on the set of simple artificial examples each less than one hundred lines of source code. All of them (after some debug) were successfully migrated to new libraries. We tested reverse migration and it was successful too.

To prove feasibility of proposed approach we decide to evaluate the developed tool on real-world project. One of the important requirements for an evaluated project was an availability of test cases which are necessary for dynamic trace extraction. We have chosen a project named *instagram-java-scraper* which uses *OkHttp* client. This project was successfully migrated too. All project tests were passing and the code review had shown that migrated code is correct.

The migration tool has an automatic test suite which checks the correctness of the migration. The test suite also performs the reverse migration tests.

7. Conclusion

In this paper, we present the results of our research in the area of a software project migration. We have developed the migration tool which uses formal library specifications for automated porting Java programs to usage of new library. We have evaluated our tool on set of artificially constructed programs and the real-world open source project. All of artificial programs and the real-world one were successfully migrated. Based on this we conclude that our approach has the right to exist.

We are planning to improve our approach and migration tool. The key directions of future research are:

- refinement of library specification formalism;
- development and extension of library model specification language;
- increasing possibilities of user control on the migration process;
- development of a more reliable and feature-rich migration tool.

References

1. Itsykson V.M. The Formalism and Language Tools for Semantics Specification of Software Libraries // Modeling and Analysis of Information Systems. — 2016. — Vol. 23, no. 6. — P. 754–766. — (In Russ.).
2. Marosi A. C., Balaton Z., Kacsuk P. GenWrapper: A generic wrapper for running legacy applications on desktop grids // 2009 IEEE International Symposium on Parallel Distributed Processing. — 2009. — May. — P. 1–6.
3. Google. A conformant OpenGL ES implementation for Windows, Mac and Linux. — 2017. — URL: https://github.com/google/angle (online; accessed: 18.05.2017).
4. QOS.ch. Simple Logging Facade for Java (SLF4J). — 2017. — URL: https://www.slf4j.org/ (online; accessed: 18.05.2017).
5. Jemerov Dmitry. Implementing refactorings in IntelliJ IDEA // Proceedings of the 2nd Workshop on Refactoring Tools / ACM. — 2008. — P. 13.
6. Cordy James R. The TXL source transformation language // Science of Computer Programming. — 2006. — Vol. 61, no. 3. — P. 190–210.
7. Stratego/XT 0.17. A language and toolset for program transformation / Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, Eelco Visser // Science of computer programming. — 2008. — Vol. 72, no. 1. — P. 52–70.
8. Baxter Ira D, Pidgeon Christopher, Mehlich Michael. DMS/spl reg: program transformations for practical scalable software evolution // Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on / IEEE. — 2004. — P. 625–634.
9. Broeksema Bertjan, Telea Alexandru. Visual support for porting large code bases // Visualizing Software for Understanding and Analysis (VISSOFT), 2011 6th IEEE International Workshop on / IEEE. — 2011. — P. 1–8.
10. Christoph Alexander, Müller Matthias M. GREAT: UML transformation tool for porting middleware applications // International Conference on the Unified Modeling Language / Springer. — 2003. — P. 18–30.
11. Transforming Code with Compositional Mappings for API-Library Switching / L. Wu, Q. Wu,

G. Liang et al. // 2015 IEEE 39th Annual Computer Software and Applications Conference. — Vol. 2. — 2015. — P. 316–325.

12. Filman Robert E, Havelund Klaus. Source-code instrumentation and quantification of events // Foundation of Aspect-Oriented Languages. — 2002. — P. 45–49.
13. JetBrains. Statically typed programming language for the JVM, Android and the browser. — 2017. — URL: https://kotlinlang.org/ (online; accessed: 18.05.2017).
14. Eisenbarth Thomas, Koschke Rainer, Vogel Gunther. Static trace extraction // Reverse Engineering, 2002. Proceedings. Ninth Working Conference on / IEEE. — 2002. — P. 128–137.