Olympiads in Informatics, 2013, Vol. 7, 123–131
© 2013 *Vilnius University*

EMAx: Software for C++ Source Code Analysis

Emil STANKOV, Mile JOVANOV, Aleksandar BOJCHEVSKI,
Ana MADEVSKA BOGDANOVA

Faculty of Computer Science and Engineering, University Ss. Cyril and Methodius
Rugjer Boshkovikj str. 16, Skopje, Macedonia
e-mail: emil.stankov@finki.ukim.mk, mile.jovanov@finki.ukim.mk,
aleksandar.bojchevski@gmail.com, ana.madevska.bogdanova@finki.ukim.mk

Abstract. Source code analysis is the process of extracting information about a program from its source code. In this paper we discuss the need for source code analysis and we present our model for a system that represents source codes as a vector of attributes. We outline the main modules of the proposed system (the parser, the executor and the parameterization module), and provide the implementation details for each module. The system that is presented has been designed to be used as a tool that provides vector representations of program solutions further used in solving the problem of source code comparison. Also, the presented system can be easily adapted and used for similar problems. It offers a unique idea for source code representation, and allows fast production of such representations.

Key words: source code analysis, source code analysis tool, programming, program evaluation.

1. Introduction

Source code analysis is an important field in computer science. Source code analysis is the process of extracting information about a program, from its source code. According to Harman (2010), the term "analysis", in this context, means any procedure that takes a source code and gives us insight into its meaning.

Considering the significant amount of code that is being produced every day, it's clear that we need more tools to analyze and understand that code. Furthermore, given the complexity of modern software, tools based on source code analysis are being increasingly used to improve performance and productivity.

Source code analysis has many applications into a variety of software engineering tasks, including: clone detection, debugging, source code optimization, source code comparison, reverse engineering, performance analysis, and many others.

Information about a program can be extracted from its source code or artifacts (e.g., from Java byte code or execution traces) generated from the source code using automatic tools (Binkley, 2007). However, given the complexity of modern software, the manual analysis of code (source code, intermediate, or machine code) is costly and ineffective. A more viable solution is to resort to tool support. Such tools provide information to programmers that can be used to coordinate their efforts and improve their overall productivity (Da Cruz *et al.*, 2009).

In this paper we present our new tool for source code analysis called EMAx. The tool has been created for the needs of a wider research in the area of source code analysis, specifically source code comparison. The paper is organized in the following manner. Section 2 briefly describes the existing models for source code analysis and illustrates one particular setting where the proposed tool can be used for evaluation of source code assessment. In Section 3 we describe the architecture and the implementation of EMAx. We outline the key modules and their specific implementations. In Section 4 we analyze the performance of the tool and describe some of its features. As a conclusion, Section 5 points out further directions for possible expanding and enhancement of its capabilities.

2. Related Work

2.1. *Existing Models for Source Code Analysis*

As stated in Binkley (2007), a system for source code analysis commonly has three components: the parser, the internal representation and the analysis of this representation. In essence, every step of the analysis converts the concrete syntax into a better abstract syntax suited to the particular analysis.

The parser parses the source code into one or more internal representations. Because of the complexities of modern programming languages, in particular those that are not LR(1), it is difficult to create an effective parser. For example, C++ is not a LR(1) language. Its grammar is ambiguous, context-dependent and potentially requires infinite look-ahead to resolve some ambiguities (Willnik, 2001).

The second major part of the model is the internal representation. There are many forms of internal representations. Some classic examples include the control-flow graph (CFG), the call graph, and the abstract syntax tree (Fischer and LeBlanc, 1988). Another popular internal representation is the static single-assignment (SSA) form, which modifies the control-flow graph in such a way that every variable is assigned exactly once, thus making def-use chains explicit (Cytron *et al.*, 1991).

The analysis of the representation can be static or dynamic. Static analysis concerns techniques for obtaining information about the possible states that a program passes through during execution, without actually running the program on specific inputs. Hence, static-analysis techniques explore a program's behavior for all possible inputs and all possible states that the program can reach (Reps *et al.*, 2004). In contrast, dynamic analysis takes into account the program's current input; however, the results are only guaranteed to be correct for the given input.

Construction of an abstract model of the program is the first step of a typical source code analysis. Clearly, it's easier to analyze a model that is not programming language specific, instead of working directly with the source code. In many cases, the first model that is created is the abstract syntax tree (AST) – a tree where each node is a construct in the source code (Kirkov and Agre, 2010). AST is usually used as a base for creating more complex graph structures (models) representing various aspects of the source code, and therefore different models are used by different source code analysis algorithms.

2.2. *Source Code Analysis in the Source Code Assessment*

Programming courses at university and high school level (especially introductory ones) often include lots of exercises in order to ease the adoption of the programming language syntax, and also to help the students to develop algorithmic way of thinking. Since programming is a compulsory course in every computer science educational curriculum, usually lots of computer science students enroll in these courses. This leads the course lecturers to the problem of mass number of solutions to exercises that have to be graded – the assessment can no longer be done manually in a reasonable amount of time.

The need for fast assessment has also been perceived in the organization of competitions in informatics. Nowadays, programming competitions require the participants to submit program source codes – solutions to concrete algorithmic problems that are given to them. Generally, the difficulty of the competition is not so much in the programming part, as it is in the design of appropriate algorithms for solving the problems at hand.

In most cases, these competitions are based on automated assessment of the submitted solutions. The automation of the assessment is necessary not only because of the large amount of solutions, but also in order to have the results in reasonable time. This is accomplished by running them on batches of input data and testing correctness of the output by comparing it to the expected output. Additionally, time and memory limits are usually enforced during this evaluation process, which allows obtaining an assessment not only in terms of the correctness of the solutions, but also in terms of their time and space complexity (Mares, 2007). Thus, the efficiency of the algorithms used is also taken into consideration.

The same or slightly similar method can be used as a solution to the previously mentioned problem of fast assessment of program codes in educational environment. There are many existing systems that are used for this purpose (Ihantola *et al.*, 2010), and the benefits are numerous, as described in Trusso *et al.* (2007).

However, the grading of the programming solutions outlined above is quite rough and strict. The grade (usually expressed in terms of number of gained points) assigned to a particular program may give a completely wrong impression about how good (and efficient) is the algorithm that it implements. As an illustration, in an extreme case, this type of automatic assessment would assign zero points to a program that, in essence, represents an implementation of a complete and 100% correct algorithm for solving the problem at hand, but uses a wrong format when printing the output data. For this reason the organizers of the international programming contests nowadays have to spend a lot of efforts to prevent such extreme cases (such as the use of feedback, and submission of multiple solutions to allow the contestants to catch these kinds of problems).

The question that we pose is the following: Is there an automated way to determine similarity of a code (the one that scored low or zero points on the grading system) to another code (that scored full score), in order to reconsider the grading of the first one?

Our wider research aims to produce an approach that can reveal a logical/semantic connection between the codes, using data mining methods, and we believe that our approach can give better results because it is not restricted by some demands as when the

task is only to detect plagiarism. Our approach consists of three main stages: (1) creating parse trees for each of the source codes under consideration, (2) extracting attributes that represent key characteristics of the source codes by calculating metrics from the obtained parse trees, and (3) applying data mining clustering methods on the dataset formed by these attribute representations in order to discover the existence of similarities among them.

The EMAx software that we present in this paper has been created for the main purpose of conducting the first two steps of our procedure for source code similarity detection.

3. The Architecture of the Proposed Tool for Source Code Analysis

EMAx was created for the purpose of extraction of attributes which are used for source code similarity detection. According to Roy and Cordy (2007), source code similarity detection algorithms can be classified as based on either: strings, tokens, parse trees, program dependency graphs (PDGs), metrics or hybrid approaches. EMAx tool takes a hybrid approach.

Parse trees are used, in the first step of building the model, because they allow for high level similarities to be detected. In the second step, metrics are calculated from the parse trees. Metrics capture "scores" of code segments according to certain criteria; for instance, "the number of different variables used" or "the number of if statements".

We can analyze the proposed model as a black box, where we have a set of source codes as input and a set of corresponding vectors of integer values as output. A high level view of the proposed model for the system is depicted in Fig. 1.

According to our observations, the architecture of the proposed system should be composed of several modules, each with input-process-output structure, where the output of one module is the input for the next one. The key modules of the proposed system are:

- Parser – this module generates an abstract syntax tree (AST) for a given source code. The output of this module gives us a "static" view of the source code. The generated AST contains every little detail about the code, including: preprocessor directives, macro expansions, comments, tree of nodes representing the syntax along the C++ grammar, names for all declarations and references, scopes, etc.
- Executor – this module takes the AST generated by the parser and builds an AST representing the execution of a source code from a given starting point. The output

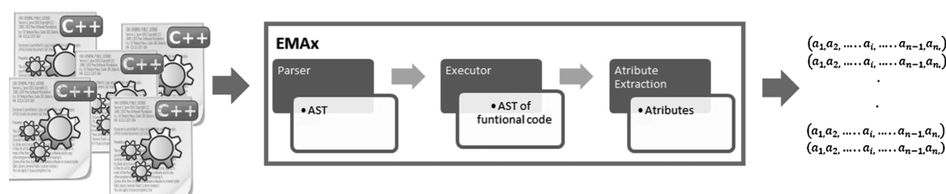

Fig. 1. High architectural view of the EMAx tool.

of this module gives us a simulated view of the possible execution of the source code. Given a starting point (usually a function), the module follows the function call hierarchy and creates the AST of the code that gets executed by the processor. It's a key module, because it provides an additional level of abstraction and bypasses the organizational decisions within a source code. It enables source code comparison on a semantic level. This means that the parts of the code that will not execute are not taken into consideration in the later analysis. This module builds an internal representation of the AST, with additional meta-data that will help the analysis.

- Attribute extraction module – this module takes the AST built by the executor and creates a vector of integer values (attributes) that represent key characteristics of the code. The attributes represent a measure for the programming structures and constructs used, as well as the relationships between them. For example, one attribute may count the number of conditional structures used (e.g., if/else statements), and another attribute may represent the relationship between a conditional structure and a loop.
- Auxiliary modules – modules for handling user interaction, displaying results, exporting data in various formats, etc.

We now describe the EMAx's implementations of each of the key modules that are outlined above. EMAx's parser uses the Eclipse CDT (C/C++ Development Tooling) parsers. CDT contains two parsers: for C and C++. These are known as DOM (Document Object Model) parsers. The parsers are compatible with the GCC suite of compilers and accept a range of GCC extensions. CDT uses the concept of a Translation Unit that represents a single source code file, as well as all the header files that are included by that file. The parser outputs an abstract syntax tree (AST) that is used as input in the next module.

EMAx's executor uses the AST generated by the CDT parser and creates an internal representation of the AST in XML format, with additional meta-data. The XML is created and later on parsed in memory to achieve better performance. The visitor looks for a user defined starting function in the original AST. Then, it follows the function call hierarchy and combines nodes from the original AST to create the new AST with additional meta-data. The meta-data include information about the current function, and track the control structures for easier parsing. Additionally, this module includes mechanisms to avoid generating potentially infinite ASTs as a result of recursive calls or circular dependencies. Furthermore, the module takes into account function calls to external libraries (e.g., the sort function in the "algorithm" library).

EMAx's parameterization module consists of two main sub-modules. The first sub-module uses the visitor design pattern defined by Eclipse CDT. The tree traversal with a visitor is depth first and can be done in both top-down and bottom-up manners. This sub-module is used to compute some of the attributes, regarding classes, inheritance, function definitions and C++ directives. The second sub-module uses the AST generated by EMAx's executor module in XML format. It employs XPath (XML Path Language) queries for computing the values of the attributes. Most of the attributes are computed by this sub-module. Both sub-modules are written in Java.

Handling the user interaction and displaying the results is done using the Eclipse Rich Client Platform (RCP). Additionally, there are modules for exporting the results of the analysis in various formats (e.g., CSV format).

EMAx has been designed to be simple and easy to use. The specification of a starting function and the selection of input source files for analysis is a simple task. The interface for loading the source codes is shown in Fig. 2.

The results from the analysis are presented in a table to the user in a minimalistic manner. The user can arrange the order of the attributes and choose a subset of the attributes that he/she is interested in. Fig. 3 shows a small part of an output produced by the tool for a set of C++ source codes given as input to it.

Fig. 2. EMAx's interface for selection of input source files.

file	for	case	if	return	switch	while	break	co...	ran...	func...	rec...	fun...	variables	uns...	array ...	arra...
575.cpp	6	0	8	1	0	1	2	0	0	1	0	0	7	1	3	0
576.cpp	8	0	11	1	0	1	3	0	0	1	0	0	7	0	3	0
610.cpp	2	0	0	1	0	0	0	0	0	1	0	0	5	0	0	0
740.cpp	3	0	2	1	0	0	0	0	0	1	0	0	9	0	0	0
741.cpp	3	0	2	1	0	0	0	0	0	1	0	0	8	0	0	0
951.cpp	2	0	2	3	0	0	0	0	0	3	0	0	5	0	1	0
1119.cpp	4	0	1	1	0	0	0	0	0	1	0	0	1	0	1	0
1120.cpp	4	0	1	1	0	0	0	0	0	1	0	0	2	0	1	0
1121.cpp	4	0	1	1	0	0	0	0	0	1	0	0	3	0	1	0
1271.cpp	3	0	0	1	0	0	0	0	0	1	0	0	3	0	2	0
1313.cpp	4	0	1	1	0	0	0	0	0	1	0	0	4	0	3	0
1314.cpp	4	0	1	1	0	0	0	0	0	1	0	0	4	0	3	0

Fig. 3. Example output of EMAx (partial view).

4. Features and Performance of the Tool for Source Code Analysis

Having in mind the motivation for creating this system (source code similarity detection), it's clear that the system must be able to handle large sets of data. Furthermore, it has to produce results as fast as possible.

As described above, one of the sub-modules of EMAx's parameterization module uses XPath queries to compute the values of the attributes. Moreover, most of the queries are fixed and don't depend on the specific source code that is being analyzed. After running a few experiments, we realized that the compilation of XPath queries took most of the processing time. This gave us an opportunity for optimization. One part of the optimization was keeping a map of pre-compiled queries, which significantly reduced (nearly in half) the processing time.

Another part of the optimization was reducing the number of traversed nodes in the original AST. Specifically, we traverse only those nodes that are of interest in the process of extraction of attributes.

In order to evaluate the performance of the system, we conducted some experiments. We took large sets of source codes from algorithmic competitions, and we tested them with our system. The results from the experiments are presented in Table 1. Each entry in this table refers to a set of source codes that represent solutions to the same problem. From the results we can conclude that the processing time is highly dependable on the number of source lines per code.

In general, the complexity of the algorithm used for generating each source code's attributes is between O(N) and O(N∗N), where N is the size of the source code in bytes. This complexity of the algorithm is a consequence of the complexities of the algorithms for creation of the parse trees, creation of the XML representation of the parse trees and evaluation of the XPath queries. Of course, the evaluation of the XPath queries on the XML representation takes up most of the processing time.

Because the system was developed in Java, using Eclipse RCP, it is able to run on various operating systems (e.g., Windows, Linux).

EMAx allows selection of a subset of attributes to be exported and analyzed which offers flexibility to any user of the tool. The system is also capable of automatic discovery

Table 1
Performance analysis of the system

Average size of source codes in SLOC (source lines of code)	Number of source files tested	Total time for completion for all source files in seconds	Average time of completion per one source file in seconds
∼ 16.2	150	∼19	∼0.127
∼ 25.3	290	∼40	∼0.137
∼ 25.1	541	∼70	∼0.130
∼ 20.17	504	∼51	∼0.101

of source codes in a directory. This feature allows easier automation of the systems that could potentially utilize this tool.

We have not found any (descriptions of) source code analysis tools that perform the same or similar function as EMAx in order to make a comparison.

5. Conclusion

In this paper we discussed the need for source code analysis. We provided an overview of systems for source code analysis, with particular focus on systems for source code comparison. Then, our model for a system that generates source code vector representations was presented. We outlined the main modules of the proposed system (the parser, the executor and the parameterization module), and we provided the implementation details for each module.

The presented system has been designed to be used as a tool that provides vector representations of source codes further used in solving the problem of source code comparison. It has been tested with real sets of source codes taken from programming competitions and has proved as very efficient in performing the task for which it is intended. Also, the presented system can be easily adapted and used for similar problems. One of the main advantages that the system offers is that it analyzes the source codes by simulating execution of the code from a given starting point. The software is highly extensible and adding support for additional languages (besides C++) is easy.

Some of our ideas for future improvement include: addition of support for extraction of user defined attributes, further optimization of the XPath queries, addition of more languages from the C/C++ family (e.g., Java), and addition of automation capabilities.

Acknowledgements. The research presented in this paper is partly supported by the Faculty of Computer Science and Engineering in Skopje.

References

Binkley, D. (2007). Source code analysis: a road map. In: *2007 Future of Software Engineering (FOSE '07)*, IEEE Computer Society, Washington, DC, USA, 2–5.

Cytron, R., Ferrante, J., Rosen, B., Wegman, M., Zadeck, K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. Syst.*, 13(4).

Da Cruz, D., Henriques, P.R., Pinto, J.S. (2009). Code analysis: past and present. In: *Proceedings of the Third International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2009)*.

Fischer, C.N., LeBlanc, R.J. (1988). Crafting a compiler. *Benjamin/Cummings Series in Computer Science*, Benjamin/Cummings Publishing Company, Menlo Park, CA.

Harman, M. (2010). Why source code analysis and manipulation will always be important. In: *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM '10)*, IEEE Computer Society, Washington, DC, USA, 7–19.

Ihantola, P., Ahoniemi, T., Karavirta, V., Seppala, O. (2010). Review of recent systems for automatic assessment of programing assignments. In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, New York, NY, ACM.

Kirkov, R., Agre, G. (2010). Source code analysis – an overview. In: *Cybernetics and Information Technology*, 10(2), 60–77.

Mares, M. (2007). Perspectives on grading systems. *Olympiads in Informatics*, 1, 124–130.

Reps, T., Sagiv, M., Wilhelm, R. (2004). Static program analysis via 3-valued logic. In: *CAV*, 15–30.

Roy, K., Cordy, J.R. (2007). A survey on software clone detection research. *School of Computing*, Queen's University, Canada.

Trusso Haley, D., Thomas, P., De Roeck, A., Petre, M. (2007). Seeing the whole picture: evaluating automated assessment systems. In: *ITALIC E-Journal of the Learning and Teaching Subject Network for Information and Computer Science (LTSN-ICS)*, 6(4), 203–24.

Willnik, E.D. (2001). Meta compilation for C++. PhD thesis, University of Surray

E. Stankov is a teaching and research assistant at the Faculty of Computer Science and Engineering, University "Ss. Cyril and Methodius", in Skopje. He is a member of the Executive Board of the Computer Society of Macedonia and has actively participated in the organization and realization of the Macedonian national competitions and olympiads in informatics since 2009. Currently he is finishing his master studies at the Faculty of Computer Science and Engineering.

M. Jovanov is a teaching and research assistant at the Faculty of Computer Science and Engineering, University "Ss. Cyril and Methodius", in Skopje. He is the president of the Computer Society of Macedonia and has actively participated in the organization and realization of the Macedonian national competitions and olympiads in informatics since 2001. Currently, he is preparing his PhD thesis on the topic of online collaborative ontology building in e-learning environment.

A. Bojchevski is an undergraduate student at the Faculty of Computer Science and Engineering, University "Ss. Cyril and Methodius", in Skopje. His areas of interest include algorithmic programming, data mining and intelligent systems. Currently, he is finishing his studies.

A. Madevska Bogdanova is an associate professor at the Faculty of Computer Science and Engineering, University "Ss. Cyril and Methodius", in Skopje. She is a member of the Computer Society of Macedonia and has participated in the organization and realization of the Macedonian national competitions and olympiads in informatics. Her research interests are in the field of intelligent systems, bioinformatics, machine learning and ICT in education. She is an author of over forty scientific papers, chapters in books in the area of computer science and presentations for popularization of informatics.