Inlining Control-Flow Jumps in Library Usage Graphs of Legacy Code

Ruben Opdebeeck
Software Languages Lab, VUB
Brussels, Belgium
ropdebee@vub.be

Johan Fabry
Raincode Labs
Brussels, Belgium
johan@raincode.com

Coen De Roover
Software Languages Lab, VUB
Brussels, Belgium
cderoove@vub.be

I. Presentation Abstract

A lot of the world today still runs on legacy code, yet a vast amount of research focuses on contemporary programming languages. For example, research involving mining usages of software libraries, such as code recommendation, API misuse detection, and library usage pattern mining, rarely considers legacy codebases, instead choosing to focus on more modern languages, like Java or JavaScript. However, for legacy code, and especially in the field of code rejuvenation, establishing a good understanding of a library and the ways in which it is used, is vital. A potential strategy to achieve this is to collect a large number of examples of library usage, and subsequently employ pattern mining to isolate those usages that frequently occur in a codebase. In this presentation abstract, we will present an approach to extract whole-program graph-based library usage models for COBOL codebases.

Various techniques to apply library usage pattern mining on codebases already exist today. Of those, extracting so-called *graph-based object usage models* (*groums*) [1] from a codebase, and using frequent subgraph mining, is a promising avenue. Such groums represent the control and data flow of a method, focusing specifically on library calls. In general, such groums contain call nodes, representing the method that is called, and data nodes, representing unique data values in the program. Control-flow edges link together different call nodes in the order in which they appear, and any branches in control flow would be represented by multiple outgoing control-flow edges in the graph. Definition and use edges link a data node to the call node that uses it as an argument, or produces it as a return value, respectively.

However, constructing groums for legacy languages such as COBOL is not a straightforward process, as there are multiple challenges to address. For example, libraries and library calls for Java programs are well-understood concepts, yet this is not the case for COBOL. We define a library call as any call to an external program which is explicitly marked as a library. Moreover, deciding at which granularity to extract such groums is a straightforward choice in OO languages, where groums often represent a single method. For COBOL, this choice is not as easy. Extracting groums at the granularity of a single paragraph would lead to too small graphs, since paragraphs are often very small units of code. Instead, we decided to extract groums for a whole program.

This brings with it a new challenge of handling inter-paragraph control flow. There are three ways that a COBOL paragraph can transfer control to another paragraph. First, under normal execution, when the end of a paragraph is reached, control passes to the next paragraph in the program. Second, when a `GOTO X` statement is encountered, control jumps to paragraph `X` and resumes normal execution from there. Last, a `PERFORM X THRU Z` statement passes control to paragraph `X` and resumes execution from there, up until the last statement of paragraph `Z`. After this last paragraph is executed, control is passed back to the statement following the `PERFORM` statement. Importantly, following a `GOTO` while a `PERFORM` is active still retains this "returning" behaviour.

The approach that we will present handles such control flow jumps using a two-phase graph construction process and graph inlining. In the first phase, we construct intermediate groums for each paragraph individually. These intermediate graphs contain auxiliary nodes to represent the points in the program where control would jump. In the second phase, we apply inlining on the intermediate graph to replace the auxiliary nodes by the graph of the paragraph that would be executed. In addition, we keep a jump table to handle the jumps that occur at the end of a performed paragraph, and a jump stack to eliminate any iteration caused by the jumps.

The groum extractor thus produces graphs that describe the control and data flow of a whole COBOL program. Such graphs can be shipped to a frequent subgraph mining algorithm, such as the one used in BigGroum [2], to infer patterns in the usage of libraries. Such patterns may prove useful in re-engineering a library to a modern programming language. Furthermore, to aid in the comprehension of complex COBOL programs, an interactive groum visualisation tool could be most useful. Such a tool is left as future work.

References

[1] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-Based Mining of Multiple Object Usage Patterns," in *Proc. 7th joint meeting of the European Software Engineering Conf. and the ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (ESEC/FSE '09)*. ACM, 2009, pp. 383–392.

[2] S. Mover, S. Sankaranarayanan, R. B. P. Olsen, and B.-Y. E. Chang, "Mining Framework Usage Graphs from App Corpora," in *Proc. 25th Int. Conf. on Software Analysis, Evolution and Reengineering (SANER '18)*. IEEE, 2018, pp. 277–289.