Mining for Graph-Based Library Usage Patterns in COBOL Systems

Ruben Opdebeeck*, Johan Fabry†, Tim Molderez*, Jonas De Bleser*, Coen De Roover*
*Vrije Universiteit Brussel, Belgium, {ropdebee,tmoldere,jdeblese,cderoove}@vub.be
†Raincode Labs, Belgium, johan@raincode.com

***Abstract*—When migrating legacy systems to more contemporary platforms, it is not always feasible to migrate their external library dependencies as well. A workaround can be to partially reimplement the library dependencies on the new platform, limiting oneself to the features that are used by the migrated system. For contemporary programming languages, several approaches to mining for library usage patterns have been proposed. For legacy programming languages, in contrast, such tools are lacking. This encumbers establishing what parts of a library need to be rewritten during a migration project, especially when considering large-scale systems.**

In this industry track paper, we present an approach to mining library usage patterns in COBOL code. We describe a library usage extractor for COBOL, which produces graphs that capture the control and data flow involved in library calls. The extractor supports legacy control flow features, such as `GO TO`. We use these graphs as input to two state-of-the-art frequent subgraph mining algorithms, and report on the scalability of their use for mining common library usage patterns in two industrial COBOL systems. The mined library usage patterns can help assess and subsequently steer the migration effort.

***Index Terms*—legacy code; software renovation; library usage pattern mining**

I. Introduction

When migrating a legacy system to a new platform, its external dependencies need to be migrated along. For COBOL, for instance, such a dependency can be a set of separate COBOL programs that are called from programs of the system being migrated. For technical or legal reasons, it might not always be possible to migrate such libraries of programs to the target platform. One possible alternative is to reimplement them partially. Indeed, only the subset of the library that is used by the system under migration needs to be reimplemented. Hence knowing how the library is used, i.e., what are the patterns of calls to the external programs and their arguments, enables assessing and planning the reimplementation effort.

It is infeasible to inspect legacy COBOL systems for library usage patterns manually. Systems under migration typically consist of thousands of COBOL files, where each file may be nontrivial in terms of the control and data flow between calls to the external programs that comprise a library. An automated means of mining for control and data flow patterns in library calls is required. Moreover, it is crucial that this mining scales to thousands of COBOL programs and potentially tens of thousands of library calls.

Fortunately, several automated approaches to library usage pattern mining have been proposed — but only for the context of more modern and object-oriented programming languages. The most notable capture the patterns in so-called *Groums* (short for Graph-based Object Usage Models), for which at least two different graph mining algorithms have been proposed [1], [2]. Moreover, the second work has shown that it scales to large corpora of software. As these algorithms take a graph of control and data dependencies as input, they should be equally applicable to legacy languages such as COBOL, provided that similar graphs can be constructed.

To enable mining of library usage patterns in COBOL, we propose a Groum extractor for COBOL. Most notably, it comprises a definition of COBOL library calls, and takes into account particularities with regard to control flow. To establish the feasibility and scalability of Groum-based mining of typical COBOL systems, we mined for usage patterns in two industrial systems: A small system of 305 programs, and a medium-size system of 3926 programs.

To the best of our knowledge, we are the first to consider Groum-based library usage pattern mining in a legacy context. Our results show that the approach is feasible, but that the large size of our Groums severely inhibits the scalability of the mining algorithms. Surprisingly, of the two approaches that have previously been proposed, the one that was shown to be more efficient [2] for contemporary OO languages, is actually less so on our corpus.

II. Background

The purpose of *library usage pattern mining* is to describe common usages of a programming library in a corpus of source code. Library usages and their patterns can be described using various representations, of which the *graph-based object usage model*, or *Groum* [1], is a prominent example. This representation describes a program unit as a graph consisting of control and data flow information that relates to the usage of OO libraries. Groums and their derivatives have previously been applied to find best practices in the Android framework [2], as well as to detect defective usages of Java libraries [3].

Figure 1 depicts an example of a Groum representing a typical usage of a Java stream. Groums use two types of nodes to show how libraries are used in a method. Call nodes, the rectangles in the figure, represent a method call. Their node labels are the called method's signature, including the method name, receiver type, and parameter types. Data nodes, depicted as ellipses in the figure, represent a unique data value in the program, and are labelled with the value's type. The

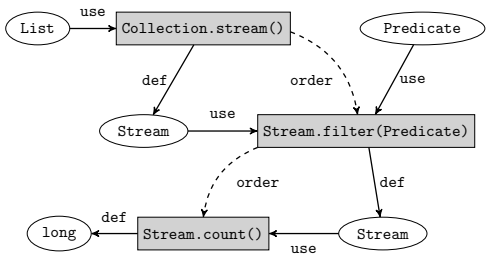

Fig. 1. An example of a Groum representing the usage of a Java stream.

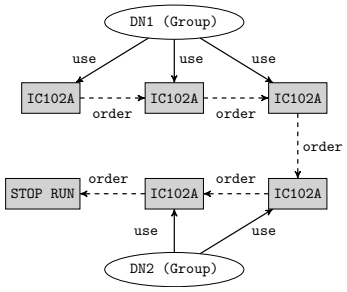

Fig. 2. Groum for the program `IC101A` of the COBOL85 test suite [4].

nodes in the graph can be connected by one of three types of edges. First, control-flow edges, dashed and labelled "`order`" in the figure, link two call nodes in the order in which they would be executed. They thus represent the control-flow graph (CFG) of the method. Second, use data-flow edges, labelled "`use`", link a data node to a call node where it is consumed, either as argument or as receiver. Third, definition data-flow edges, labelled "`def`", link a call node to the data it produces. Formally, Groums also contain transitive control-flow edges, representing the transitive closure over the CFG. For clarity and brevity of the figures, we omit these here.

Uncovering library usage patterns in a corpus of OO programs proceeds according to two phases. First, in the extraction phase, a Groum is created for each method. Then, in the mining phase, the Groums are subjected to a frequent subgraph mining algorithm to discover popular subgraphs, which are considered patterns. The algorithms use a *support* parameter, which specifies how often a subgraph needs to occur in the corpus of Groums for it to be considered popular, and thus, a pattern.

Two algorithms have been proposed to this end. The first, by Nguyen et al. [1], is an apriori pattern-growth algorithm. It continually generates candidate patterns as extensions of previously-generated frequent patterns, until the support of the candidate pattern becomes too low. To calculate the support of a candidate extension, it needs to be compared to all other candidates for isomorphism. Since isomorphism checking for graphs is expensive, the algorithm approximates these checks using a vector-based approximation. The second algorithm, used in BigGroum [2], first partitions the corpus of Groums using frequent itemset mining on the call labels used in each Groum. It then slices each of the Groums in a partition to remove nodes that are irrelevant to the method calls in the partition. Finally, it decides whether a sliced Groum is frequent by calculating how often it appears as a subgraph of the sliced Groums in the partition. The subgraph checks are encoded as SAT formulae, enabling BigGroum to outperform GrouMiner on large corpora of Groums [2].

As Groums essentially contain only control and data flow information, and are not inherently limited to OO programs, it should be possible to represent COBOL library usages as Groums. Furthermore, given the promising results reported by Mover et al. on BigGroum [2], their back-end may be able to scale to the size of corpora we wish to mine. We hence developed a compatible Groum extractor for COBOL source code, so that we can reuse this back-end to mine for library usage patterns in COBOL. We present our extractor next.

III. Extracting Groums from COBOL source code

Groum mining has been developed for object-oriented languages, which precludes its straightforward transposition to non-OO (legacy) languages like COBOL. We describe the challenges encountered while extracting Groums from COBOL programs, and provide an overview of our solutions.

Figure 2 depicts a Groum extracted from the `IC101A` program from the UK National Computing Centre Cobol-85 compliance test suite [4]. It shows a sequence of five calls, where the first three use a variable named `DN1` as the argument, while the last two use `DN2`. The last node of the Groum indicates that the program terminates through the `STOP RUN` statement.

a) Definition of library usages: Since Groums describe library usages, it is necessary to define what constitutes a library. In languages that support packages, this is straightforward. For example, a user could mark a package as a library, and any call to a method defined in that package could be considered part of a usage of that library.

As COBOL does not have support for packages, this strategy is not applicable. Instead, we allow the user to specify which COBOL programs should be treated as library programs. Thus, interactions with a library take the form of a `CALL` statement into a library program. Such statements are modelled as call nodes in our Groums, labelled with the name of the called program. Alternatively, the user can specify that *all* `CALL` statements should be considered library invocations. This is useful when it is not known upfront which programs are part of a library.

COBOL also provides the `CANCEL` statement, which resets the state of an external program and is thus important for library usages. We also model such statements as call nodes, but prefix their label with "`CANCEL`".

Another complicating factor is that COBOL's `CALL` and `CANCEL` statements can take the name of the program in a variable, rather than as a string literal. Rather than attempting to determine the run-time value of this variable through a data flow analysis, we choose to encode such instances as a combination of the variable name and its initial value. This allows us to consider such cases during pattern mining,

while also distinguishing between the two formats. Of course, call destinations computed through later assignments to the variable might be missed in this manner.

b) Control flow: The default control flow in COBOL is to execute the program statements in lexical order, but control can also jump to different places in the program. To allow this, COBOL programs can consist of multiple *sections* and *paragraphs*. For the purpose of this text, we can consider these as labelled sequences of statements, and use the term "paragraph" to mean "paragraph or section".

The control flow of a program can jump between paragraphs in two ways. First, a statement of the form `PERFORM A THRU Z` will jump to a paragraph named `A` and sequentially execute all of the subsequent statements up to, and including, the statements of paragraph `Z`. When control flow reaches the end of `Z`, the program execution jumps back to the statement following the original `PERFORM` statement. Second, a statement of the form `GO TO X` jumps to a paragraph named `X`, where it continues sequentially. These two types of jumps can be mixed arbitrarily. For example, if a `GO TO` statement is executed while a `PERFORM` statement is 'active', and control eventually reaches the boundary of the `PERFORM`, control will then be passed back to the statement after the `PERFORM` statement. This allows for complex control flow jumps during the execution of a COBOL program, which is not possible in languages for which Groums were designed.

c) Inter-paragraph Groums: Paragraphs can be fairly small, and in those cases constructing a Groum for each paragraph individually does not sufficiently describe the program's control flow. Therefore, we need to construct "inter-paragraph" Groums that represent the control flow of the whole program, and thus need to account for inter-paragraph control flow jumps. We do this using a two-phase construction approach.

First, we create intra-paragraph Groums that contain auxiliary nodes to indicate the presence of `GO TO` or `PERFORM` statements and the target labels to which they jump. These intra-paragraph Groums also contain an implicit jump node as their last node, whose target depends on whether or not a `PERFORM` statement is active.

Second, we perform *graph inlining* on these auxiliary jump nodes, i.e., we replace each of these nodes by the paragraph(s) that it would execute. When a `PERFORM` node is encountered, the implicit end-of-paragraph jump node of its last paragraph is adjusted to jump back to the node following the `PERFORM`. Thus, the inliner can straightforwardly replace any jump node by the paragraph to which is jumped, and recursively inline the inlined subgraph.

d) Iterative control flow: Programs may execute the same statements repeatedly through iteration. However, since Groums are DAGs, it is impossible to add a back-edge to represent this. We therefore represent iterative control flow structures by approximating them as executing the statements at most once, which is equivalent to the strategy employed for Java Groums [1].

Inter-paragraph control flow jumps are another source of potential iteration, e.g., a `GO TO` statement may jump back to a paragraph that was executed before, which would eventually reach the same statement once again. We detect such cases by keeping track of each inlined paragraph for each path down the Groum. When the inliner encounters a jump to a paragraph that was already inlined in this path, it prevents any further inlining down that path, yet continues to inline down other paths. Note that we cannot simply skip the jump and proceed to inline the subsequent statements. Since a `GO TO` statement does not return control, we do not know which statements these may be.

e) Program termination: COBOL offers multiple statements that terminate a program, e.g., `STOP RUN` or `GOBACK`. Additionally, certain programs may terminate the caller program, akin to Java's `System.exit`. We therefore allow the user to specify a set of termination-inducing programs. When the extractor encounters a statement that would lead to program termination, it halts the construction of that path in the Groum. Both types of program termination are represented as call nodes. This enables representing how a program terminates, while retaining compatibility with BigGroum's backend.

f) Data flow information: Groums also contain data flow information related to library calls. This includes arguments used in the invocations, as well as data returned by the library call. However, program calls in COBOL typically do not return data, rather, they mutate values in the memory space shared between the caller and callee. Consequently, determining which data is "returned" by a program call constitutes running a data flow analysis on the called program. This may be impossible for the purpose of library re-engineering, where the source code of the called program may not be available. Thus, we do not extract data definitions from the library calls, and the Groums only contain data usages.

g) Language support tradeoffs: COBOL has a very rich and complex syntax, yet we do support Groum extraction for all programs that we can successfully compile with the Raincode Labs compiler. This is done by focusing on statements that have an inherent effect on control flow. Statements that do not relate to control flow (e.g., `MOVE` to assign a variable) are handled generically, by traversing their constituents in search of relevant statement types.

We handle all forms of conditionals, iterative structures, and control flow jumps, except for the `ALTER` statement. This statement can be used to dynamically change a label to point to another paragraph, and thus may affect the target of a `GO TO` or `PERFORM` statement. Although this statement could be supported through a data flow analysis, we believe it would significantly complicate the extractor for limited gains. Thus, support for this statement is left for future work.

IV. Evaluation

To validate that we are able to mine graph-based library usage patterns for large COBOL codebases in reasonable time, we devised two experiments. We measured the time taken to extract and mine patterns in two industrial COBOL codebases from Raincode Labs clients.

	# Programs	# KLOC	Min. support
Case 1	305	662.2	30
Case 2	3926	22889	400

TABLE I
SUMMARY OF THE TWO INDUSTRIAL COBOL CODEBASES.

Targeting industrial COBOL codebases, we posit the following research questions:

RQ1 Does the COBOL Groum extractor scale?

RQ2 Can the extracted Groums be mined for patterns in a reasonable time?

A. Experimental Setup

To answer these research questions, we selected two industrial COBOL codebases. Their properties are described in Table I. The first case is a system of 305 COBOL programs and the second case is nearly 4000 COBOL programs. We chose a minimum support threshold of around 10% for each.

We have run our COBOL Groum extractor on both of these cases, measuring the time it takes to construct Groums from their abstract syntax trees. The extractor was configured to consider all `CALL` statements to be library calls. We then fed the constructed Groums into the two Groum mining algorithms described in Section II. We reused the BigGroum algorithm implementation provided by Mover et al. [2] and we implemented an adaptation of the GrouMiner algorithm [1]. The latter is based on the version presented by Amann et al. [3], which is itself based on GrouMiner's implementation. Our custom GrouMiner algorithm contains a number of optimisations, mainly intended to reduce its memory footprint to allow it to mine larger graphs. However, the fundamentals of the algorithm remain the same. To emphasise the difference between the original implementation and ours, we will refer to our implementation as GrouMiner*.

Recall that the support of a potential pattern is the number of unique Groums in which it occurs as a subgraph. The minimum support value for pattern mining in both cases was set to roughly 10% of the programs in the case. For BigGroum, we used the same support value for Groum partitioning as for pattern mining. Lastly, we instructed both pattern miners to produce patterns that contain at least 2 call nodes.

All experiments were performed on a machine with 32GB RAM and a 6-core, 2.6GHz CPU. For GrouMiner*, the maximum JVM heap space is set to 32GB, whereas BigGroum was allowed to use all available memory. All of the timing values were obtained from sequential implementations.

B. RQ1: Is the COBOL Groum Extractor Scalable?

Table II summarises the results of our first experiment. We measured the full time taken to extract COBOL Groums, including the construction and inlining phases. The table also lists the average size and maximum size of the extracted Groums, which includes all its nodes and edges.

These results show that our Groum extractor for COBOL can indeed efficiently handle both small and medium-sized COBOL codebases. For the small codebase, extraction takes less than a minute, whereas for the medium-sized project, extraction takes roughly 17 minutes. Although these figures do not scale linearly with the number of programs, we note that the extracted Groums are significantly larger in the medium-sized case. Thus, we believe that our extractor will be able to scale to large COBOL codebases.

Case	Time (s)	# Groums	Avg. size	Max. size
Case 1	43.8	273	244	4634
Case 2 (full)	1019	3925	4769	1217422
Case 2 (limited)	N/A	3573	125	9644

TABLE II
RESULT OF EXTRACTING GROUMS ON THE TWO CODEBASES.

Corpus	GrouMiner*		BigGroum		
	Time (min)	# P	Time (min)	# Part.	# P
Corpus 1	0.95	3	10.4	3	4
Corpus 2	121.8	7	618.2	1	0

TABLE III
RESULT OF MINING GROUMS FROM THE CORPORA.

It is worth noting that the size of the Groums we extract significantly differs from the size of the Groums extracted for contemporary OO languages. For example, even for the small case, our average graph size is larger than the largest Groum of any case studied by Nguyen et al. [1]. This is a natural consequence of the fact that our Groums are extracted for whole COBOL programs, whereas OO Groums are extracted for a single method. It is also interesting to highlight the largest Groum extracted from the medium-size codebase. The Groum contains over 16.000 nodes, and more than a million edges, a graph size unseen by any of the two mining algorithms. Although this may appear to be an abnormally large graph, it can be expected that the interplay of inter-paragraph control flow jumps and conditional control in complex COBOL programs produces graphs of such sizes.

Such large graph sizes may have a significant effect on the performance of a Groum miner. Preliminary experiments showed that neither of the two algorithms is able to handle the medium-sized codebase on our machine. BigGroum crashed after two hours with an overflow error in the SAT solver, whereas GrouMiner* quickly exhausted the allotted 32GB of JVM heap space. Therefore, we derived a new Groum corpus for the second case, depicted in the third row of Table II. In this new corpus, we discarded any Groum that has more than 100 call nodes, significantly reducing the average size of the graphs while only slightly lowering the number of Groums in the corpus. We will thus perform Groum mining only on the corpus of the first case, and the reduced corpus of the second.

C. RQ2: Can the Groums be Mined in Reasonable Time?

Table III depicts the results of mining the extracted Groums with the GrouMiner* and BigGroum algorithms. The column labelled "# P" indicates the number of patterns that were discovered. For BigGroum, the column labelled "# Part." contains the number of partitions it generated.

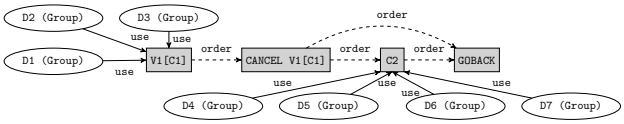

Fig. 3. An example of a pattern extracted by GrouMiner.

We find that GrouMiner* outperformed BigGroum on both corpora by a large margin. Moreover, the patterns produced by GrouMiner* are generally more descriptive than BigGroum's patterns. For example, in the first case, BigGroum produced patterns that consist of few call nodes, with no edges relating them to one another. In the second case, BigGroum produced no patterns, even though it ran for over 10 hours.

Figure 3 depicts an anonymised version of a pattern extracted by GrouMiner* from the first corpus of Groums. This pattern succinctly shows that the COBOL programs in the codebase often call and afterwards reset another program through a variable named `V1`, whose initial value is `C1`. Afterwards, they optionally call a program `C2` directly, and terminate via the `GOBACK` statement. This pattern has a support of 79 instances, and thus occurs in nearly 30% of the programs. The `V1` call is responsible for a particular user interface screen, specified through the provided argument variable. For subsequent uses, this program needs to be reset to its initial state after use, hence requiring the `CANCEL` call. The `C2` program is a generic database invocation routine that is configured through its different arguments.

This experiment demonstrates that the size of the extracted Groums has a significant impact on the scalability of the pattern mining algorithms, as is to be expected. Perhaps surprisingly, we also find that on our codebases BigGroum scales worse than GrouMiner*. We hypothesise potential causes for this observation in the next section.

V. Discussion

In Section IV-C, we found that BigGroum fails to outperform GrouMiner* in both of our case studies. This is a surprising observation given that BigGroum has previously been shown to scale better than GrouMiner on large corpora of Groums [2].

Closer inspection of the second corpus reveals that among the more than 60.000 call nodes in the corpus, there are less than 15 unique labels. Moreover, most of the Groums in this corpus contain all of these labels. This means that when BigGroum partitions the Groums into clusters, it creates a single cluster for all of these labels. Consequently, when it slices the Groums, the Groums only decrease in size marginally, since most of the nodes are relevant to the cluster. As a result, BigGroum has to check many large graphs for subgraph isomorphism, which involves expensive SAT solver calls. Indeed, BigGroum spent roughly half of its mining time (5.2h) inside of the SAT solver.

In contrast, the GrouMiner and GrouMiner* algorithms do not have to check for subgraph isomorphism of such large graphs. Instead, they only have to check for graph isomorphism of smaller candidate pattern graphs. On the other hand, they often have to check more graphs for isomorphism, as they may generate several potential candidate extensions. Moreover, the pattern growth strategy employed allows them to find common subgraphs in these large graphs. BigGroum cannot find these, since it requires that Groum slicing reduces the graphs sufficiently to make the sliced graphs themselves frequent. Since slicing does not reduce the size of our Groums significantly, this does not happen.

We suspect that for BigGroum to be efficient, the corpus of Groums needs to either contain relatively small graphs, or contain a large and varied set of call labels. For contemporary OO languages, where library usage is prevalent and Groums can be extracted for single methods, both of these conditions likely hold. However, for COBOL Groums, neither assumption holds. Because our Groums are extracted for a whole program, they tend to be very large. Moreover, as is evident from the small set of call labels, COBOL programs use only a small set of libraries. We conclude that the BigGroum algorithm is not applicable to Groums that have properties similar to ours: large graphs or a small set of call labels.

VI. Conclusion

In this industry track paper, we have presented an approach to mining graph-based library usage patterns in COBOL codebases. The mined patterns can be used to assess and steer the effort of reimplementing legacy dependencies of a system under migration. The approach can be instantiated with the frequent subgraph mining algorithms that have proven themselves in the context of OO software.

To evaluate the scalability of our approach, we compare two such instantiations on a COBOL codebase of 660 KLOC and on one of 22.9 MLOC. The results demonstrate the feasibility of our approach, but also that the large graph sizes render mining expensive. Surprisingly, the best-performing algorithm for OO library usage patterns performs the worst when applied to COBOL library usage.

Acknowledgements

This research was partially funded by the Belgian Innoviris TeamUp project INTiMALS (2017-TEAM-UP-7).

References

[1] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-Based Mining of Multiple Object Usage Patterns. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '09)*, pages 383–392. ACM, 2009.

[2] Sergio Mover, Sriram Sankaranarayanan, Rhys Braginton Pettee Olsen, and Bor-Yuh Evan Chang. Mining Framework Usage Graphs from App Corpora. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER '18)*, pages 277–289. IEEE, 2018.

[3] Sven Amann, Hoan Ahn Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. Investigating Next-Steps in Static API-Misuse Detection. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19)*, pages 265–275. IEEE, 2019.

[4] National Computing Centre, UK. COBOL85 test suite. https://www.itl.nist.gov/div897/ctg/cobol_form.htm.