Towards Classification of Loop Idioms Automatically Extracted from Legacy Systems

Joji Okada*, Takashi Ishio†, Yuji Sakata*, and Katsuro Inoue‡
*System Engineering HQ, NTT Data Corporation, Tokyo, Japan
†Graduate School of Science and Technology, Nara Institute of Science and Technology, Nara, Japan
‡Graduate School of Information Science and Technology, Osaka University, Osaka, Japan
Email: okadaju@nttdata.co.jp, ishio@is.naist.jp, sakatayu@nttdata.co.jp, inoue@ist.osaka-u.ac.jp

***Abstract*—Legacy systems are important in business but difficult to maintain. One of the causes of the difficulties is a large number of code clones in the systems; Those clones implement similar functionalities using common loop idioms in a company. Since the loop idioms have been developed to implement popular functionalities, most of them are likely to be translated into simple SQL statements in a new, modernized version of a system. To investigate the feasibility of the approach, we propose a method to automatically extract cloned loop idioms embedded in COBOL program files. We manually classified the extracted idioms and labeled them according to their functionalities. We evaluated the accuracy of our classification result with three experts.**

***Index Terms*—Legacy Migration, Reverse Engineering, Program Comprehension, Clustering, COBOL**

I. Introduction

There are still many enterprise systems on mainframes (hereinafter referred to as legacy systems) that are important in business but difficult to maintain [1]. In order to respond to business changes, companies need to keep legacy systems updated. Rebuilding a legacy system using modern programming languages and execution environments is an important activity to reduce the future maintainance cost [2].

In legacy systems, many clones have almost the same control structure of loops and conditional branches (hereinafter referred to as loop idiom). Fig. 1 shows two example code fragments. Both of them have a WRITE statement in an IF statement in a LOOP statement and a READ statement outside the IF statement. They select a subset of rows from an input file and write the result into an output file. In a rebuild process, developers replace those programs with SELECT statements in SQL. In a software company, expert developers often recognize the functionality of a program (i.e. a type of SQL statement to replace the program) by such a loop idiom used in the program. Based on the observation, we hypothesize that two programs implement the same functionality if they have the same loop idiom. By classifying program files into functional groups using loop idioms, we enable developers to efficiently rewrite similar programs with similar SQL statements.

In this paper, we propose a method to automatically extract cloned loop idioms from COBOL program files. We extract frequent loop idioms from program files and manually label them as a dicitionary of loop idioms. The dictionary enables developers to obtain a functionality label for a program file.

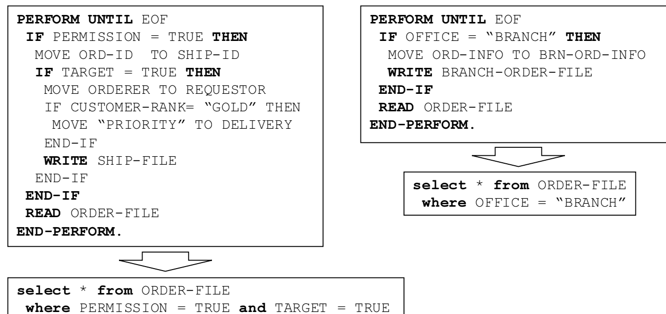

Fig. 1. Example clones in legacy systems

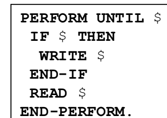

Fig. 2. A loop idiom extracted from the left program in Fig. 1

Those loop idioms representing the same functionality can be considered as a kind of Type-4 clones.

To evaluate the accuracy of the method, we built a dictionary of loop idioms in two legacy systems. We compared the classification result of 100 samples with a manual result created by three experts. As a result, 81% of the samples are correctly labeled using the dictionary.

II. Proposed method

Our method focuses on COBOL programs in a batch processing system. A program reads input from one or more files and writes some processed result to one or more files. We extract a loop idiom involved in a program through the following two steps.

First, we extract only the following statements using a COBOL parser as the loop idiom of a file: (1) Input statement (e.g. READ), (2) Output statement (e.g. WRITE), (3) Loop statement enclosing input/output statements, and (4) Branch statement enclosing input/output statements. We remove subroutine calls from a file by inlining the contents of the called routines.

Next, we apply the following normalization rules to ignore programming style differences.

- Replace identifiers and constants with anonymous tokens
- Replace nested branch statements with a single branch statement

TABLE I
PROFILE OF TARGET LEGACY SYSTEMS

System	Domain	# All Files	LOC	# I/O Files	LOC
A	Insurance	3,051	2.3M	1,165	1.1M
B	Finance	10,442	14.8M	7,988	13.3M

TABLE II
LABEL OF PROGRAM FILE

Label	Meaning
Filter	Output only a subset of rows that match an extraction condition.
Split	Split the content of an input file into multiple files depending on conditions.
Grouping	Classify rows into groups and calculate values for each group, e.g. min/max values.
Match-Filter	Compare the contents of two input files and write the result to a single file.
Match-Split	Compare the contents of two input files and write the result to separated files depending on conditions.
Union	Copy all rows of two input files to a single output file.
Report	Insert additional lines such as headers and footers.
Edit	Modify all lines of an input file.
Dump	Generate a file without any input file.
Load	Read a file and generate no output files.
Multiple	Perform multiple functionalities.

- Replace consecutive IF-ELSE-IF statements having three or more control-flow paths with a SWITCH statement
- Replace a repeated sequence of the same statements with a single occurrence of the statements

The normalization steps translate a file into a loop idiom. Fig. 2 shows a resultant idiom extracted from the left program in Fig. 1. We classify files having the same loop idiom into the same functionality. After that, we manually label the loop idioms based on their functionalities. Different loop idioms may have a common functionality label.

III. Evaluation

To investigate the feasibility of the method, we built a dictionary of loop idioms from two actual legacy systems. The extraction of the loop idioms was automated while the labeling was manually performed. During the labeling process, we refer to only loop idioms. We did not refer to the contents of program files.

Table I shows an overview of the two systems. Those systems are still actively maintained in a software company. The column # I/O Files indicates the numbers of program files including at least one input/output statement. The remaining program files include subroutines of the systems.

We have obtained 11 functionality labels from loop idioms. Table II shows the labels and their meaning. In Table III, the columns of System A and B show the numbers of files and loop idioms for each functionality label in two systems. The top 10% loop idioms cover 50% of program files.

To evaluate the accuracy of the dictionary, we applied our dictionary-based labeling to sample program files. The ground truth has been manually created by three experts; we explained the functionality labels and their meaning, and then each expert individually labeled programs. The experts decided answers for each program by voting and discussion.

We extracted 100 program files from System A using the stratified sampling method due to the limited time of experts.

TABLE III
RESULTANT LABELS (F=FILES, I=IDIOMS CLASSIFIED BY THE AUTHORS)

Label	System A		System B		Accuracy (System A)		
	#F	#I	#F	#I	#F	#correct	
Filter	144	8	809	10	15	12	(80%)
Split	119	7	369	8	13	13	(100%)
Grouping	9	3	226	5	1	1	(100%)
MatchFilter	56	9	473	15	6	6	(100%)
MatchSplit	2	1	49	1	1	1	(100%)
Union	0	0	57	1	0	0	(N/A)
Report	56	5	24	5	6	6	(100%)
Edit	123	7	1,093	8	14	13	(93%)
Dump	52	1	366	1	5	0	(0%)
Load	42	1	1,048	1	4	1	(25%)
Multiple	562	402	3,474	2,359	35	28	(80%)
Total	1,165	444	7,988	2,414	100	81	(81%)

The number corresponds to a confidence level of 95% and a sample error of 10%. The accuracy columns in Table III show the numbers of sample files and correctly classified files by the dictionary. For example, eight Filter idioms appear in 144 files of System A, we extracted 15 files from the 144 files randomely. Further 12 files are corrected. The accuracy is higher than 80%. However, experts classified program files having Dump and Load idioms into other groups such as Filter and Report. This is because those program files included DB access statements that were not taken into consideration in our normalization rules.

IV. Related Work

Balachandran [3] proposed a code search algorithm using a structural similarity of abstract syntax trees. It enables a user to search code examples using syntactic patterns ignoring semantic differences such as data types and function names in a query code fragment. Our approach takes a similar approach to ignore semantic differences but uses heuristics tailored for legacy systems.

Allamanis et al. [4] proposed a method of pattern mining focusing on loops in the source code. While this method focuses on variable reading and writing, our method deals with file I/O in a system.

V. Conculsion

We proposed a method to extract clones like loop idiom, identify the functionality of a COBOL program based on its loop idiom, and applied the method to actual legacy systems. In the future work, we would like to extend the method to support database I/O and perform an experiment.

References

[1] Khadka, R., Batlajery, B. V., Saeidi, A. M., Jansen, S. and Hage, J.: How do professionals perceive legacy systems and software modernization?, Proceedings of the 36th International Conference on Software Engineering, pp. 36–47, 2014.

[2] Ganesan, A. S. and Chithralekha, T.: A Survey on Survey of Migration of Legacy Systems, Proceedings of the International Conference on Informatics and Analytics, pp. 72:1–72:10, 2016.

[3] Balachandran, V.: Query-by-example in large-scale code repositories, U.S. Patent No. 9,317,260. 2016.

[4] Allamanis, M., Barr, E. T., Bird, C, Devanbu, P., Marron, M. and Sutton, C. Mining Semantic Loop Idioms, IEEE Transactions on Software Engineering. vol. 44, no. 7, pp. 651–668, 2018.