Analysis of Industrial Control System Software to Detect Semantic Clones

Jnanamurthy HK
School of Electrical Engineering
& Computer Science
University of Newcastle
Australia
jnanamurthy.hogavanaghattakumarswamy@uon.edu.au

Raoul Jetley
Software Research Group
ABB Corporate Research Center
India
raoul.jetley@in.abb.com

Frans Henskens
School of Electrical Engineering
& Computer Science
University of Newcastle
Australia
frans.henskens@newcastle.edu.au

David Paul
School of Science & Technology
University of New England
Australia
dpaul4@une.edu.au

Mark Wallis
School of Electrical Engineering
& Computer Science
University of Newcastle
Australia
mark.wallis@newcastle.edu.au

Sudarsan S D
Software Research Group
ABB Corporate Research Center
India
sudarsan.sd@in.abb.com

***Abstract*—The detection of software clones is gaining more attention due to the advantages it can bring to software maintenance. Clone detection helps in code optimization (code present in multiple locations can be updated and optimized once), bug detection (discovering bugs that are copied to various locations in the code), and analysis of re-used code in software systems. There are several approaches to detect clones at the code level, but existing methods do not address the issue of clone detection in the PLC-based IEC 61131-3 languages. In this paper, we present a novel approach to detect clones in PLC-based IEC 61131-3 software using semantic-based analysis. For the semantic analysis, we use I/O based dependency analysis to detect PLC program clones. Our approach is a semantic-based technique to identify clones, making it feasible even for large code bases. Further, experiments indicate that the proposed method is successful in identifying software clones.**

***Keywords*- Software Clones, PLC programs, Software Maintenance, Software-Reusability.**

I. Introduction

Maintenance costs of large software systems are high in part due to a significant amount of duplicated code present in it [1][2]. Thus, costs can be reduced if there are techniques available to detect software clones. There is research to find cloned code automatically for software re-usability, code optimization, plagiarism detection, extracting program features and software bug detection [1][2][3]. Baxter et al. [3] define a software clone as a piece of code that is structurally or semantically similar to another part of a code. Definitions of clones carry some indefiniteness, and this often causes a clone detection process to be much more complicated than the detection method itself [2].

There are several reasons to influence the software developer to generate software clones [3][4][5], and clones can also be introduced by accident [6][7][8]. Copying existing code, with or without modification is a simple form of reuse procedure which results in software but is frequently used by developers as a cost-cutting technique [9]. Another approach of re-using strategy is code forking [10]; forking re-uses similar solutions that have diverged with the evolution of the software system. For example, when creating a new software system, re-using code of a similar project with slight modifications as per the requirements. Because it can significantly reduce production time, reuse of functionality and logic from existing projects is often a good thing, especially when the features are implemented in a library; though often code is simply copied.

Several surveys [11] show that 5-20% of the code is similar in most software applications. One of the significant shortcomings of copied code fragments is that bugs present in one particular code fragment can be copied to duplicated code fragments requiring each bug to be independently identified and corrected. Re-organizing and re-using code is another issue in software maintenance that causes repeated bugs [12]. Importantly, re-using bug-free code fragments is gaining more importance to reduce development time and cost in same domain applications [13].

A wide range of techniques and tools are available to detect clones for high-level languages such as C, C++ and Java. These tools offer limited support for PLC-based IEC 61131-3 languages due to the change in syntactic and semantics of the language. In this paper, we propose a method to detect semantic clones in PLC-based IEC 61131-3 languages based on I/O variable impact and dependency analysis.

The rest of this paper is organized as follows. The necessary background and literature required for the proposed method are defined in Section 2 and Section 3. A solution with algorithms for semantic clone detection is described in Section 4. Then our results and discussions on a real-time system are

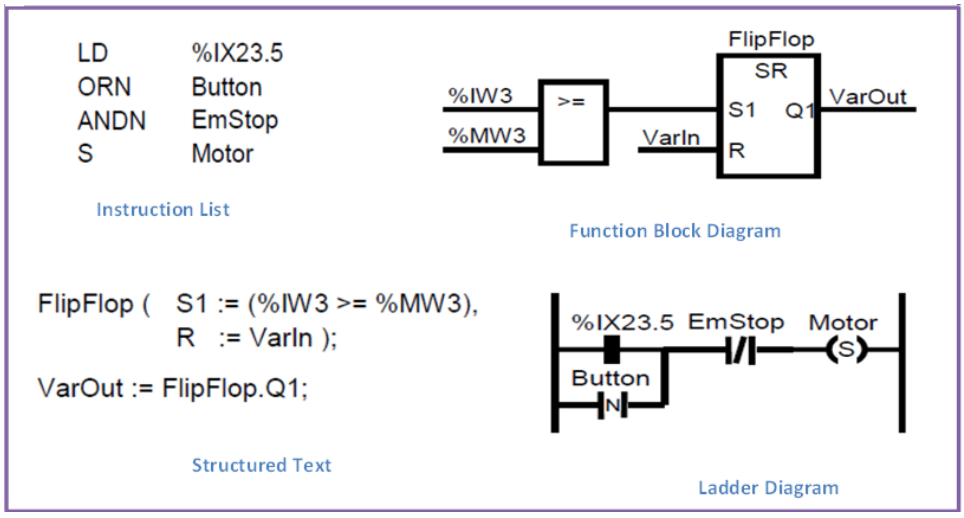

Fig. 1: PLC-based IEC 61131-3 Languages

presented in Section 5. Finally, Section 6 concludes the paper.

II. Background

This section provides the necessary information and background required for our proposed semantic-based clone detection methods.

A. Clone Taxonomy

A taxonomy helps to understand the definition of software clones. Most existing and typical definitions [11] of clones are:

Code Fragment: A contiguous segment of code from a software system.

Clone: A pair of code fragments that are similar, either semantically or syntactically.

Type 1: Code fragments that are syntactically identical, except for differences in white space and comments.

Type 2: Code fragments that are syntactically similar, except for a difference in identifier names and literal values.

Type 3: Code fragments that are syntactically similar with differences at the statement level.

Type 4: Code fragments that are semantically similar, but syntactically different.

B. Overview of PLC-based IEC 61131-3 Languages

PLC-based IEC 61131-3 [14] describes Programmable Logic Controller (PLC) programming languages and also provides the concept and guidelines to create PLC projects. A PLC is a computer used to control a machine and associated processes. PLCs are designed to control industrial control systems and to be very flexible in how they interface with inputs and outputs to the real world. A PLC has two main components: 1) The Controller and 2) The input/output section. The most important unit of PLC is a CPU module. The CPU consists of a microprocessor, memory and other circuits to control logic, monitoring and communications. I/O devices consist of digital or analog devices. A digital I/O card handles discrete devices which provide a signal that is either on or off such as push-buttons and sensors.

In PLC-based IEC 61131-3 languages, Program Organization Units (POUs) are considered as *Blocks* in a conventional programming language. POUs are the independent software units in an application and can be called with or without parameters. There are three types of POUs: Function, Function Block and a Program. Function and Function Block always produce the same result for the same input, but the Function does not have any memory to store data records. A Function block has memory to store a data record and status information. The Program represents a user's PLC program which can access I/O of the PLC.

Variable declaration in a POU consists of three parts: Input variable declaration, Output variable declaration and Local variable declaration. Input variables are the input signals to POUs and Output variables are the return values of a POU. Any other variable that belongs to a particular POU is considered as a Local variable. A POU is typically programmed using one of the following programming languages:

1) Structured Text (ST): ST is a well-structured language that supports a range of constructs for functions, assignments, expressions, conditional statements etc. ST is familiar due to features like its compact, clear code layout and well defined logical plan.
2) Instruction List (IL): Instruction list is a traditional PLC language, and it is similar to simple machine assembly language.
3) Ladder Diagram (LD): Ladder diagrams are schematic diagrams commonly used in industrial control systems. Ladder diagrams resemble a ladder with two vertical rails to supply power and horizontal lines as rungs to represent a control circuit.
4) Functional Block Diagram (FBD): Function block diagrams are a graphical and straightforward way to program any functions together in a PLC program.

A simple fragment of code to give examples of the PLC-based IEC 61131-3 languages is shown in figure 1 [15].

III. Literature

Several tools [16] exist to detect clones. CCFinder [17] is a multilinguistic token-based software clone detection system. CCFinder transforms source code into tokens for comparison; this method uses metrics to analyze the clones. Clone detective [18] is a framework that helps to configure clone detection processes. Ctcompare [19] is a tool to detect type 1 and type 2 clones. Ctcompare is a tool to lexically analyze the code and produce a sequence of tokens. Then the sequence of tokens is broken into overlapping tuples. Tuples are then hashed and the hashed value is used to detect clones. Iclones [20] is a framework for incremental analysis to detect clones. Iclone is considered as a revision approach to identify clones (Analysis of previously analyzed document). DECKARD [21] is a tree-based approach to detect clones, where sub-trees are clustered based on the similarity distance calculated in Euclidean space. DECKARD provides an environment to identify type 1 and type 2 clones. SourcererCC [22] is a tool to detect type 2 and type 3 clones. SourcererCC is based on inverted-index to discover clones and SourcererCC filters the comparisons required to detect the clones by analyzing token positions.

During the study of these tools, we found that they are all based on token handling and tree-based analysis to detect either syntactic or semantic clones but not the both. These tools and clone detection techniques are also not defined to identify clones in PLC-based IEC 61131-3 languages. IEC 61131-3 languages are different in structure and organization of code as compared to other high-level languages.

IV. Semantic Clone Detection in PLC-based IEC 61131-3 Language

In this section, we define algorithms to find semantic clones using I/O variable dependency analysis as summarized in figure 2. The proposed semantic analysis is more effective to identify syntactic differences and reordered statements. Our approach uses semantic information to identify code clones rather than syntactic structural information. Semantic information may be control information, data type information, data dependencies or similar information. To detect semantic clones in ST programs, we developed algorithms based on Output variable dependency-based analysis and Input variable impact dependency. The proposed method is used with the PLC-based IEC 61131-3 language Structure Text (ST) at two different levels: 1) Output variable dependency slicing analysis 2) Input variable impact slicing analysis. In our method, we analyze two types of dependency. Let Input be all inputs to a POU and Output be all outputs of the POU, with $\{in_1, in_2, in_3, .., in_n\} \in Input$, $\{out_1, out_2, out_3, .., out_n\} \in Output$, then the function $f_n(in_1, in_2, in_3, .., in_n) = out_n$, represents the relationship between I/Os. The following dependencies can then be defined:

1) Output variable depends on input.

$$\forall_{\{out\} \in Output}, \exists_{\{in_1, in_2\} \in Input}(out \longleftarrow \{in_1, in_2\})$$

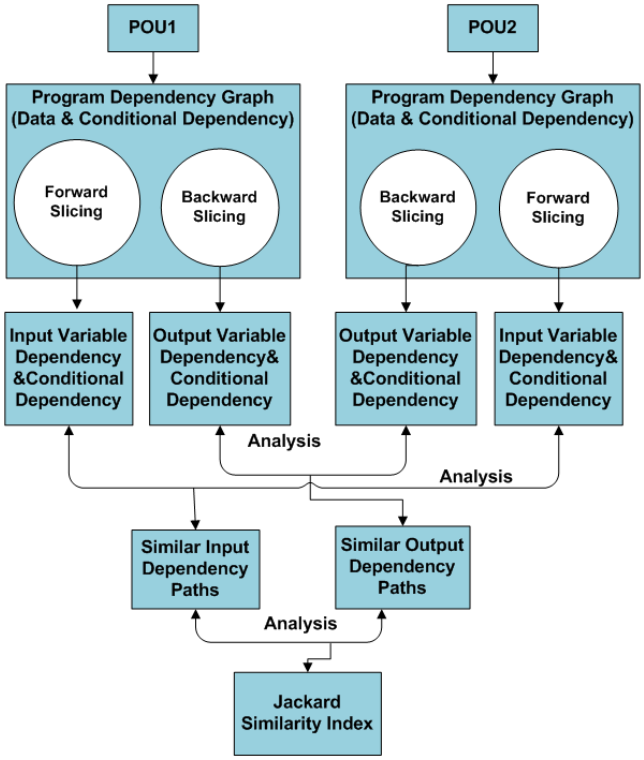

Fig. 2: I/O variable dependency analysis

2) Output variable depends on variables.

$$\forall_{\{out\} \in Output}, \exists_{\{v_1, v_2\} \in Variable}(out \longleftarrow \{v_1, v_2\})$$

where $\longleftarrow$ is a variable data dependency relation and 'Variable' is the set of variables present in the POU.

A. Output Variable Dependency Slicing Analysis

Analyzing output variables semantically in IEC 61131-3 language leads to an efficient way to detect code clones rather than considering all the variables present in the POUs; this efficiency is due to the importance of I/O information in the controller applications. Analyzing I/Os of controller code reduces complexity by excluding unnecessary analysis of all variables in the code. To create output variable dependencies, we execute the following steps:

- construction of AST
- extraction of statements
- assigning of unique identification number
- extraction of control statements
- extraction of data dependency
- creation of PDG
- output variable dependency slicing.

We used Abstract Syntax Tree (AST) of ST programs [23] to extract statements for the purpose of Program Dependence Graph (PDG) creation. Another important necessity to create a PDG is assigning a unique number to each statement to allow identification of each statement's location. Algorithm

Algorithm 1 Variable_Dependency(Dictionary <int, ParseTreeNode> ListNodes, Conditions)

```
for ( v=ListNodes.length, v ≥ 0, v–) do
    if (ListNodes[v] of type V^def ∈Variable_Definitions) then
        ∃_{U∈v_dependents}, ListNodes[v]^variable_def ⟵U, where ⟵ be the dependency relation
        dependent_variables=parse(U)
        for (d ∈ dependent_variables ) do
            for ( i= v-1, i ≥ 0, i–) do
                if (ListNodes[i] of type V^def ∈Variable_Definitions&& d==ListNodes[i]^variable_def ) then
                    dependency_definition_location ←i;
                    for (con ∈ Conditions) do
                        if (i >= con.Start && i <= con.End) then
                            CondiStore.add(con)
                        end if
                    end for
                    Add_dependency(d, dependency_definition_location, CondiStore)
                end if
            end for
        end for
    end if
end for
```

1 shows the sequence of steps to create program variable dependency. ListNodes of type Dictionary<int, ParseTreeNode> and Conditions of type Dictionary<int, int, ParseTreeNode> are the input parameters to algorithm 1. ListNodes consist of a unique statement number of type integer and program statements in the form AST. Further, Conditions are control statements information in a POU of the form <Start, End, Condition>, where 'Start' and 'End' is the beginning and ending point for each condition 'Condition'. Furthermore in the algorithm 1, the parse() is a method to extract dependent variables (A ← B, A is a definition part and B is a dependent part consists set of variables to define A). PDG is used to extract output variable definitions dependency slices; PDG slices are the semantic information which contain dependent data for each output variable defined in the programs. Later, output variable slices are used to analyze between two different POUs using sequential matching methods.

Definition 1: Let $\{out_1, out_2, out_3, ..., out_n\} \in POU$ be the output variables belonging to each program organization unit. $\forall out \in POU$, 'out' can be defined 'j' $(0 < j < n)$ times based on the program organization unit requirements. Output variable dependency slices out^{slices} can then be defined as

$$\forall_j, \{\pi^j_{out} \longleftarrow \pi^{j-1} \longleftarrow \pi^{j-2} \longleftarrow \pi^{j-3}, .., \pi^{j-n}\} \in out^{slices}$$

where π^j_{out} is the output variable at 'j' position, π^{j-1} be the dependent variables defined at 'j-1' position, 'out' denotes output variable and $\longleftarrow$ represents the variable dependency relation. Common output variable slices out^{slices}_{common} is an intersection between output variable slices belong to different POUs.

$$\forall_{out1^{slices} \in POU_1}, \forall_{out2^{slices} \in POU_2}, then$$
$$\{out^{slices}_{common} := out1^{slices} \cap out2^{slices}\}$$

To analyze between output variable dependency slices, we considered the following features:

- Dependency among Data types, Direction, FD port, Initial Value, Variable attributes of POUs.
- Conditions to hold in the dependency slices.

B. Input Variable Impact Dependency Slicing Analysis

Analyzing input variables semantically leads to an efficient way to detect code clones and reduces complexity in the analysis by excluding unnecessary analysis of program data (considered input variable impact dependency). To create input variable impact dependencies, we execute the following steps:

- construction of AST
- extraction of statements
- assigning of unique identification number
- extraction of control statements
- extraction of forward impact dependency
- creation of PDG
- slicing of input variable impact dependency.

The analysis steps for Construction of AST, Statements Extraction, Assigning unique identification number and Control statements extractions are similar to the Output variable dependency slicing analysis section. The program structure is a combination of data dependence and control dependence; in this case, the PDG encodes semantic information such as variable impact dependency and control dependency uniformly. Algorithm 2 describes the variable impact dependency extraction process, which is used for analysis of input variables in each POUs.

Definition 2: Let $\{in_1, in_2, in_3, ..., in_n\} \in POU$ be the input variables belonging to each program organization unit. $\forall_{in \in POU}$, 'in' can be used 'j' $(0 < j < n)$ times based on the program organization unit requirements. Input variable impact

Algorithm 2 Variable_Impact_Dependency(Dictionary <int, ParseTreeNode> ListNodes, Conditions)

```
for ( v=0, v ≤ ListNodes.length, v++) do
    if (ListNodes[v] of type V^{def} ∈Variable_Definitions) then
        ∃_{U∈v_dependents}, ListNodes[v]^{variable_def} ⟵U, where ⟵ be the dependency relation
        for ( i← v+1, i ≤ ListNodes.length, i++) do
            ∃_{U'∈v_dependents}, ListNodes[i]^{variable_def} ⟵ U'
            dependent_variables=parse(U')
            Var=ListNodes[v]^{variable_def}
            if Var∈dependent_variables && Var ∉ Variable_redefined) then
                dependency_definition_location ←i;
                for (con ∈ Conditions) do
                    if (i >= con.Start && i <= con.End) then
                        CondiStore.add(con)
                    end if
                end for
                Add_dependency(d, dependency_definition_location, CondiStore)
            end if
        end for
    end if
end for
```

dependency $Input^{slices}$ slices defines as

$$\forall_j, \{\pi^j_{in} \longrightarrow \pi^{j+1} \longrightarrow \pi^{j+2} \longrightarrow \pi^{j+3}, .., \pi^{j+n}\} \in Input^{slices}$$

π^j_{in} is the input variable at 'j' position, π^{j+1} is the input variable impact at j+1 position and $\longrightarrow$ represents the impact relation. Common input variable impact dependent slices In^{slices}_{common} as an intersection between input variable impact slices belong to different POUs.

$\forall_{Input1^{slices} \in POU_1}, \forall_{Input2^{slices} \in POU_2}$, then

$$\{In^{slices}_{common} := Input1^{slices} \cap Input2^{slices}\}$$

V. Results and Discussion

Here we present the details of the experiment and results considering both Output variable dependency slicing analysis and Input variable impact slicing analysis.

A. Experiment Setup

The work described in this paper conducted with real-world industrial applications from a large tech company in the domain of industrial automation, robotics and electrical equipment. We analyzed the machine control software system implemented in PLC-based IEC 61131-3 language Structured Text (ST). We considered libraries of a control software system with a size threshold of minimum 1k LOC (line of code) at the time of the study. We executed the proposed framework for ST programming language using the following configuration. For each library, we set the framework to generate clones using five randomly selected POUs with a minimum of 1k LOC. We considered all Code-blocks as one unit with an order (order is an execution order of Code-blocks) specified by the POU.

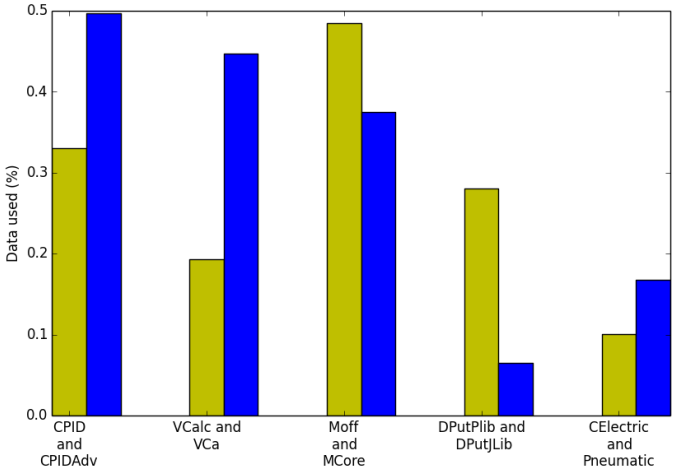

Fig. 3: Data used for Input/Output dependency analysis

B. Experiments

We created sample data for 500 instances of type-3 and type-4 I/O dependency and impact clone samples with the help of domain experts in ST language for analysis. We extracted AST for the programs, and we used the AST to create PDGs. Table I represents the analysis of POU's (CPID and CPIDAdv) output variable dependent slices. In this table columns 1 and 3 represent the output variable names belonging to different POUs, columns 2 and 4 represents the number of program slices for each output variable defined in each POU, column 5 represents similar output variable slices between the POUs and column 6 represents the Jaccard similarity index, which indicates a similarity value among the POUs dependency slices. Consider the parameters in the first row of table I

TABLE I: Output variable dependency slices analysis

POU1's Output variable name	No. times output variable defined	POU2's Output variable name	No. times output variable defined	Similar slices count	Jaccard Co-efficient
IPar	26	Interaction	48	22	0.42
BParrd	8	Pard	8	4	0.33
IPar	26	OutPar	1	0	0.00
IPar	26	HCmd	22	14	0.41
FeedOut	6	IPar	48	0	0.00
IPar	87	NameP	5	0	0.00
FeedFOut	6	FOut	1	0	0.00
FeedFOut	6	HCmd	22	0	0.00
IOP	21	InP	5	2	0.08
OutP	1	FOut	6	0	0.00
OutP	1	Outr	1	1	1.00
OutP	1	HCmd	22	0	0.00
HCmd	19	IPar	48	8	0.13
HCmd	19	FOut	6	3	0.13
HCmd	19	FOut	1	0	0.00
HCmd	19	HCmdP	22	14	0.51

TABLE II: Recall Measurements

Tool	**Intra-Library Clones**		**Inter-Library Clones**	
	Type-3	Type-4	Type-3	Type-4
SemClone	100%	97%	100%	97%
SMachine	97%	94%	97%	94%
SourcererCC	99%	0%	86%	0%
CCFinderX	70%	0%	53%	1%
Deckard	76%	1%	46%	1%
iClones	84%	0%	78%	0%
NiCad	100%	0%	100%	0%

with variable name IPar that belongs to CPID with output variable definition dependency slices having a value of 26 and the variable named Interaction that belongs to CPIDAdv with output variable definition dependency slices of 48. In the 5th column, the value 22 represents similar output variable slices detected, and 0.4230 in a 6th column represents Jaccard similarity index among POUs. Jaccard similarity index lies between 0 and 1, with a value towards 1 representing similarity increased. In row 3, the Jaccard similarity index is 0 indicating that IPar and OutPar are not similar.

We extracted 500 features from both output dependency and input impact dependency. Then we analyzed these features with the sample data and calculated a recall for extracted features. Recall results for various detectors, as compiled from the source [24] are given in Table II. CCfinder [17], NiCad [25], SourcererCC [22], Deckard [21] and iClones [20] results indicated a 53-70%, 100%, 86-99%, 46-76% and 78-84% success rate in detecting type-3 clones, respectively. SMachine [24] is a tool based on machine learning to detect type-4 clones, obtained a recall of 94%. Proposed method 'SemClone' detects semantic clones with a recall of 97% for type-4 clones and 100% for type-3 clones. We conclude that our approach improves the detection of semantic clones in PLC-based IEC 61131-3 Languages.

Figure 3 shows the amount of data (percentage of PDG dependency slices) used by SemClone to analyze the POUs to detect semantic similarity among the POUs. The first bar of figure 3 shows data used to detect semantic similarity across CPID and CPIDAdv POUs and indicated that only 33% and 49.7% of data was required to analyze CPID and CPIDAdv POUs. This shows SemClone's success at reducing the complexity and thus the time required to analyze large POUs and enhances the performance of the analysis to detect code clones. The other columns show that these results are fairly typical of the tool. Analyzing I/O in POUs helps to detect code clones effectively and reduces complexity by reducing the analysis of complete program variables. On testing on the proposed method, we observed that less than half of the data was required for analysis to detect code clones in PLC programs.

VI. Conclusions

In this paper, we present a novel approach to detect clones in the PLC-based IEC 61131-3 language using semantic-based analysis. For the semantic analysis, we used I/O based dependency analysis to detect PLC program clones. We used output variable dependency-based analysis and input variable impact usage analysis due to the importance of input and output variables in PLC programs (inputs and outputs are the signals which are essential to the operation of a controller). Analyzing inputs and outputs variable dependency rather than all data variables present in the program result in a great reduction in the amount of data that needs to be processed to detect software clones. Our approach is a semantic-based technique to identify clones and performed better than existing clone-detection methods to detect Type 3 and Type 4 software clones.

References

[1] O. I. Al-Sanjary, A. A. Ahmed, A. A. B. Jaharadak, M. A. M. Ali, and H. M. Zangana, "Detection clone an object movement using an optical flow approach," in *2018 IEEE Symposium on Computer Applications Industrial Electronics (ISCAIE)*, April 2018, pp. 388–394.

[2] M. R. H. Misu and K. Sakib, "Interface driven code clone detection," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, Dec 2017, pp. 747–748.

[3] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 368–. [Online]. Available: http://dl.acm.org/citation.cfm?id=850947.853341

[4] J. Akram, Z. Shi, M. Mumtaz, and P. Luo, "Droidcc: A scalable clone detection approach for android applications to detect similarity at source code level," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 01, July 2018, pp. 100–105.

[5] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, Jul 2002.

[6] V. Kfer, S. Wagner, and R. Koschke, "Are there functionally similar code clones in practice?" in *2018 IEEE 12th International Workshop on Software Clones (IWSC)*, March 2018, pp. 2–8.

[7] C. K. Roy and J. R. Cordy, "Benchmarks for software clone detection: A ten-year retrospective," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2018, pp. 26–37.

[8] H. Jnanamurthy, F. Henskens, D. Paul, and M. Wallis, "Clone detection in model-based development using formal methods to enhance performance in software development," in *2018 3rd International Conference for Convergence in Technology (I2CT)*, April 2018, pp. 1–8.

[9] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in oopl," in *Empirical Software Engineering, 2004. ISESE '04. Proceedings. 2004 International Symposium on*, Aug 2004, pp. 83–92.

[10] C. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful," in *2006 13th Working Conference on Reverse Engineering*, Oct 2006, pp. 19–28.

[11] M. Mondal, C. K. Roy, and K. A. Schneider, "Bug propagation through code cloning: An empirical study," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2017, pp. 227–237.

[12] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 256–256.

[13] M. Mondai, C. K. Roy, and K. A. Schneider, "Micro-clones in evolving software," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2018, pp. 50–60.

[14] R. Ramanathan, "The iec 61131-3 programming languages features for industrial control systems," in *2014 World Automation Congress (WAC)*, Aug 2014, pp. 598–603.

[15] M. T. Karl-Heinz John, in *IEC 61131-3: Programming Industrial Automation Systems*, 2010, pp. VI, 390.

[16] J. Svajlenko and C. K. Roy, "Evaluating modern clone detection tools," in *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 321–330. [Online]. Available: http://dx.doi.org/10.1109/ICSME.2014.54

[17] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, Jul 2002.

[18] E. Juergens, F. Deissenboeck, and B. Hummel, "Clonedetective - a workbench for clone detection research," in *2009 IEEE 31st International Conference on Software Engineering*, May 2009, pp. 603–606.

[19] W. Toomey, "Ctcompare: Code clone detection using hashed token sequences," in *2012 6th International Workshop on Software Clones (IWSC)*, June 2012, pp. 92–93.

[20] N. Gde and R. Koschke, "Incremental clone detection," in *2009 13th European Conference on Software Maintenance and Reengineering*, March 2009, pp. 219–228.

[21] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *29th International Conference on Software Engineering (ICSE'07)*, May 2007, pp. 96–105.

[22] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 1157–1168. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884877

[23] S. Nair, R. Jetley, A. Nair, and S. Hauck-Stattelmann, "A static code analysis tool for control system software," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2015, pp. 459–463.

[24] A. Sheneamer and J. Kalita, "Semantic clone detection using machine learning," in *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, Dec 2016, pp. 1024–1028.

[25] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *2008 16th IEEE International Conference on Program Comprehension*, June 2008, pp. 172–181.