Exploring Code Clones in Programmable Logic Controller Software

Hannes Thaller*, Rudolf Ramler†, Josef Pichler† and Alexander Egyed*
*Institute for Software Systems Engineering, Johannes Kepler University Linz, Austria
Email: hannes.thaller@jku.at, alexander.egyed@jku.at
†Software Competence Center Hagenberg GmbH, Austria
Email: rudolf.ramler@scch.at, josef.pichler@scch.at

***Abstract*—The reuse of code fragments by copying and pasting is widely practiced in software development and results in code clones. Cloning is considered an anti-pattern as it negatively affects program correctness and increases maintenance efforts. Programmable Logic Controller (PLC) software is no exception in the code clone discussion as reuse in development and maintenance is frequently achieved through copy, paste, and modification. Even though the presence of code clones may not necessary be a problem per se, it is important to detect, track and manage clones as the software system evolves. Unfortunately, tool support for clone detection and management is not commonly available for PLC software systems or limited to generic tools with a reduced set of features. In this paper, we investigate code clones in a real-world PLC software system based on IEC 61131-3 Structured Text and C/C++. We extended a widely used tool for clone detection with normalization support. Furthermore, we evaluated the different types and natures of code clones in the studied system and their relevance for refactoring. Results shed light on the applicability and usefulness of clone detection in the context of industrial automation systems and it demonstrates the benefit of adapting detection and management tools for IEC 611313-3 languages.**

1. Introduction

The increasing share of software for Programmable Logic Controllers (PLCs) and its practical importance have recently been acknowledged by the authors of the 2016 ranking of programming languages in IEEE Spectrum [1]. They found that languages for PLCs are on the rise, although in contrast to general-purpose languages such as C, C++ or Java, they specialize in a niche. Yet their "relative popularity indicates just how big that niche really is" [1]. With the growing importance of PLC software, an increasing demand for software engineering best practices and tool support is essential. The focus of this paper is on detecting and analyzing code clones in PLC programs.

Code clones are source code fragments that have been duplicated for reuse, e.g., by copying and pasting [2]. Code clones have the reputation to negatively affect program correctness [3] and to increase maintenance efforts [2]. This form of reuse is widely considered an anti-pattern in software development [4] and clones are treated as a bad smell in code [5]. Recent studies have shown that there are various reasons why code clones are introduced and that the presence of clones is not per se a problem. However, the ability to detect, track and manage clones as the software system evolves is of the essence in successful software development [6].

PLC systems are no exception in the discussion of clones as reuse in the industrial automation domain is often achieved through cloning and modifying of existing software systems or sub-systems [7]. This is caused by technological restrictions introduced by programming languages such as the lack of inheritance or polymorphism, organizational limits like time constraints, or simply by the system's complexity. Furthermore, cloning is often used as a lightweight software product line strategy to cope with various hardware options and application environments.

A wide range of tools and techniques for clone detection and management is available for programming languages such as C, C++ or Java [8]. Similar support for IEC 61131-3 languages is mostly limited to general purpose tools, which lack analysis features that require the interpretation of the syntactical structure of the analyzed language. The contributions of this paper are as follows:

- Quantitative results from a code clone analysis in a real-world PLC software system consisting of IEC 61131-3 Structured Text (ST) and C/C++.
- The extension of the widely used clone detector Simian [9] with language support for Structured Text.
- A comprehensive study on the relevance and natures of the found clones.
- An assessment of whether language support in clone detectors is of importance or not.

To the best of our knowledge, this paper presents the first study on clone detection for Structured Text. It sheds light on the questions about the applicability and usefulness of clone detection in the context of industrial automation systems and discusses the need for adapting tools for analyzing programs based on IEC 61131-3 languages.

Section 2 describes the background and related work on clone detection. The industry context is presented in Section 3, the results of the code clone analysis in Section 4. Section 5 presents the study on the relevance of found clones, their nature and empirical results on whether detectors should be adapted to support IEC 61131-3 languages.

2. Background and Related Work

Duplicating code fragments during software development activities has a long history. Definitions, taxonomies,

978-1-5090-6505-9/17/$31.00 ©2015 IEEE

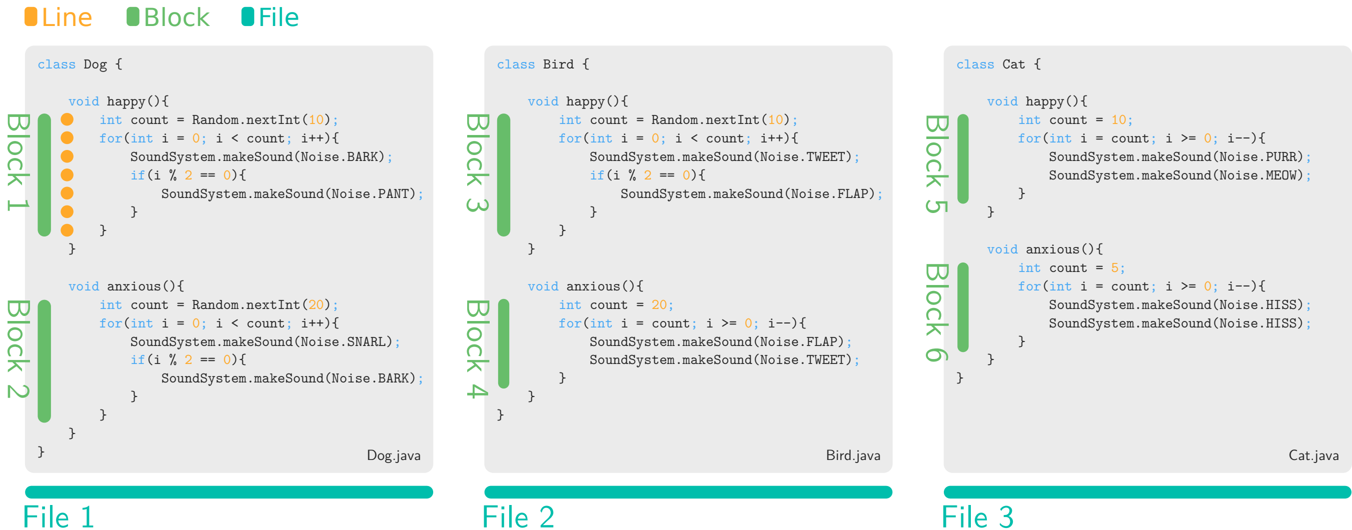

FIGURE 1: Clones may be within a file but also between files and all similar blocks form a clone class (e.g., Block 1-3).

tools for detecting, analyzing, visualizing and managing code clones exist for several languages and technologies. The interest in code clones is also reflected in the wealth of existing research and the widespread use of tools and techniques in quality management and continuous integration.

Clone pairs and *clone classes* [2] are basic terms used in the context of clone detection. A clone pair describes two code blocks, also called fragments, that are equal according to a similarity operator. A clone class is the set of all blocks that are equal according to a similarity operator, effectively forming an equivalence class. Figure 1 shows an example for a clone pair formed by Block 1 and Block 2, where the similarity operator ignores literals and constants. Examples for clones classes are Block 1-3 and Block 4-6, as they all contain equivalent blocks.

Despite these basic notions, no single holistic definition exists for code clones. This is due to the different tools and their associated publications that redefine code clones according to the capabilities of the respective tool. The issue is that code clones are intuitively well-understood, but hard to formalize such that a clear and consistent definition, that covers all applications, has not been found yet. This study relies on the definition by Baxter et al. [10] as it abstracts detection method specifics without being too vague: *Code clones are segments of code that are similar according to some definition of similarity.*

2.1. Clone Taxonomy

A taxonomy based approach helps to align the understanding of code clones, complementary to existing definitions. The typical and most frequently used categorization of code clones [2], [8], [11] is:

Type 1 (Exact Clones): Program fragments that are identical except for variations in whitespace and comments.

Type 2 (Parameterized Clones): Program fragments that are structurally/syntactically similar except for changes in identifiers, literals, types, layout and comments.

Type 3 (Near-Miss Clones): Program fragments that include insertions or deletions in addition to changes in identifiers, literals, types and layouts.

Type 4 (Semantic Clones): Program fragments that are functionally similar (i.e. perform the same computation) without textual similarity.

These types yield basic insights into the vague term of *similarity* in the code clone definition. Code fragments can be similar based on their textual representation (Type 1-3) or can have similar functionality without textual similarities (Type 4). Type 1 and 2 clones are the focus in this paper. Code clone types also characterize the accepted difference between code fragments participating in a clone, and they further define capability levels of detection tools.

2.2. Clone Tools

Clone detection tools can be categorized into detection, analysis and management tools, which are often integrated into quality management platforms. Detection tools find code clones; the results are then filtered, visualized and categorized by means of analysis tools. Management tools track existing clones and their evolution to make them an integral part of the quality management process.

Detection tools can be basically categorized into text, token, tree, graph, metrics and model-based tools or hybrid approaches [8], [11]. Text-based detectors use string-matching algorithms to find similar source code parts. Token-based methods leverage lexical analysis to extract token sequences fed into a suffix-tree/array to discover clones. Tree-based tools expose the abstract syntax tree to apply tree similarity algorithms. Each approach has advantages and disadvantages that often limit their capabilities in detecting certain clone types. Text-based tools can only detect Type 1 clones and by using language dependent normalizations their capabilities improve up to Type 3 clones. Tree-based tools use computational intensive algorithms but can detect clones up to Type 3. To summarize, detection tools are the basis of clone detection and differ in their algorithmic interpretation

of source code, which ultimately affects their capabilities. A detailed overview of detection tools is given by Bellon et al. [8], Koschke [12], and Rattan et al. [11].

Clone analysis is concerned with filtration, visualization, and categorization of clones and is often tightly coupled with clone management. Common visualizations are tree maps and scatter plots [13]–[17], but also parallel plots [18] are used. A tree map uses interactive tiles that reflect the directory hierarchy colored according to their duplication intensity. Scatter plots enumerate files along the x and y-axis where each data point reflects a duplication relationship. The combination of both provides insight into the clone relationship between but also the clone intensity within files. Filtering and ranking of clones help to organize the typically large result sets of detectors. This is done manually in conjunction with visualizations or automatically based on metrics and predefined criteria. For instance, Gemini [16] or CLICS [19], [20] are tools that make use of metrics and filtering criteria to provide the most relevant subset of clones. Categorization sorts clones into views such that the inspection can be focused on a specific task. These views may be related to the location (*Same File*, *Same Directory*, etc.), the region (*Function to Function Clones*, *Macro Clones*, etc.), or the block classification (*Initialization Clones*, *Loop Clones*, etc.) [19] of the code fragments.

Management tools help to track the clones such that they can be actively incorporated into quality assessments and architectural decision processes, but also to evaluate their evolution. This is especially important with the increased demand of modularization in the machine and plant industry [21]. Clones are often deliberately introduced as light weight variability mechanism, hence they exceed typically one product life-cycle. One way to manage these clones is, for example, CloneTracker [22]. It builds a model of the tracked clones and provides notifications if cloned code is changed or edited simultaneously. Another example is ECCO, Extraction and Composition for Clone-and-Own [23]. It uses fork clones in conjunction with a feature model to build a proper Software Product Line (SPL). This allows active reuse of fork clones as they are transformed into a well-defined corpus of reusable and combinable modules.

2.3. Related Work

Code clones are well investigated by the research community resulting in a good understanding why source code is copied. Roy and Cordy [2] presented a comprehensive overview of reasons for cloning extracted from various publications. For instance, Kim et al. [24] conducted an ethnographic study on the code clone behavior of software developers by recording the file changes. Not only language limitations forced the developers to copy code, but developers actively used the copy and paste history to determine the abstractions within their system. Another example, given by Kapser and Godfrey [25], describes several different forking patterns in which large proportions of the system are copied in order to enable software ports, specific hardware implementations or (experimental) variants.

These reasons indicate – in contrast to the incentive earlier publications give [3], [10], [13], [26]–[30] – that code clones are not universally bad or result of unskilled programmers. In fact, follow-up publications found quite the contrary [24], [25], [31]–[37], as Rattan [11] reported, especially with respect to the stability and faults caused by code clones. Possible advantages of clones during development activities are risk avoidance [2], architectural improvements [25], performance improvements (e.g., loop unfolding, reduced call overhead) and improved code stability [32], [34], [36]. Interestingly many found advantages also show up as disadvantage indicating that measuring the impact of clones is a non-trivial task. Concluding, it is clear that it depends on more than just whether code is duplicated to make statements about the quality of a system. Nevertheless, awareness and suitable methods to track and process clones are recommended, so that benefits of cloning can be leveraged and drawbacks can be mitigated.

3. Industry Context

The work described in this paper was conducted with our industry partner, a large high-tech company in the domain of machinery for metal processing. Together we analyzed a pre-release version of a machine control software system. It consisted of modules implemented in the IEC 61131-3 language Structured Text and modules written in the C/C++ programming language. The total size of the software system was 191 kLOC (Lines Of Code, LOC) with 157 kLOC in ST and 34 kLOC in C/C++, at the time the study was conducted. These numbers include only the code authored by our industry partner.

The software system was part of a large industry project and had already been evolved over several iterations with an overall development history of more than two years. In each iteration, major functional extensions were integrated, tested and stabilized. Furthermore, every iteration also included extensions that added support for different machine types and hardware variants. It was expected that this evolution led to code clones, as existing software routines were reused for similar hardware options by following a simple forking approach. Hence, code fragments up to entire subsystems were copied from the existing implementation to support the requirements of the new machine types or hardware variants.

4. Code Clone Analysis

We analyzed the PLC software system with Simian [9], a proprietary text-based clone detector, and evaluated a subset of the found clones. Simian can detect Type 1 clones in all text sources but incorporates additional normalization features for several common programming languages. These normalization features were re-implemented for Structured Text such that all languages used in the studied system (ST and C/C++) could be analyzed on the same capability level, i.e., Type 2 clones.

The analyzed source code has in total 99 538 C/C++ and 160 132 ST significant (non-whitespace) lines distributed over 372 C/C++ and 770 ST files. This includes C/C++ and ST libraries as header files. The source base contains multiple variants of the system for the different machine types, consequently, large portions of the code are very similar, leading to many clones. Many of these clones are deliberately introduced and manually managed to simplify

TABLE 1: Results of the Simian clone detection on the entirety of the code base including libraries and definition files.

Language	Option	Files with Clones	Duplicated Lines	Duplicated Blocks	Total Files	Total Sig. Lines
C/C++	Default	257	58,741	1,510	372	99,538
	Identifier	334	99,620	4,005		
	Literal	274	62,051	1,828		
	Identifier/Literal	340	117,080	4,776		
ST	Default	552	43,697	4,591	770	160,132
	Identifier	633	105,787	10,930		
	Literal	558	57,557	5,179		
	Identifier/Literal	650	133,488	12,291		

Clone overlap is allowed Minimum number of lines = 5

the product line aspect of the development process. Table 1 contains the number of duplicated lines, blocks, and files found by the detector. The clone analysis did allow for clone overlaps in order to find partial copies of variant files while simultaneously allowing full copies. The number of duplicated lines is strongly dependent on the minimum number of lines a clone is allowed to have, which was 5 lines throughout the study. This setting is already fairly low but was chosen with the subsequent study in mind.

5. Code Clone Study

A group of experts inspected clones found during the clone analysis (Section 4) within the system of our industry partner (Section 3). These inspections evaluated the nature (type) of the clones as well as their relevance for refactoring in order to help to answer the following questions: 1) What natures of clones exist within the system?, 2) Is there a difference between the natures between C/C++ and ST?, 3) How does the tool support influence the relevance of clones?, and 4) How does the clone selection approach influence the relevance of clones?

Ultimately, these questions provide the first incentive for developers of PLC software to adapt existing clone detection tools for IEC 61131-3 languages such as Structured Text.

5.1. Study Design

A subset of the clone detector results was selected and presented to a group of experts. The experts inspected the clones and provided an evaluation of each clone with respect to their nature and relevance. The responses were then analyzed and used to explore the relationships and occurrences of clones, tool support, selection approach, and languages.

5.1.1. Controlled Variables. The usage of clone detection tools raises some typical questions related to the tool configuration (normalizations, minimal line length, etc.), as well to the selection approach used when analyzing the discovered clones. Each answer potentially changes the type of clones and their perceived relevance. Therefore it is important to understand the impact of these variables when managing clones in development and maintenance projects. This study controls for the variables programming language, tool option, and clone selection approach.

- **Language:** This variable refers to the used programming languages and captures the heterogeneity of the system's code base, which reflects a typical setup in which multiple technologies are used in concert to solve a complex problem. The used languages are *C/C++* and *Structured Text* (ST). C/C++ are widespread general purpose programming languages for "system-near" applications. ST is a high-level block structured language designed for PLCs defined by the IEC 61131-3 standard. Both languages are procedural and imperative. They exhibit basic similarities but nonetheless they differ in syntax and expressiveness.
- **Option:** Simian offers basic analysis capabilities that can detect Type 1 clones in any text source. In addition it provides normalization features for several popular programming languages to support the detection of Type 2 clones. In this study the following combinations of options were used: *Text* (source code is interpreted as normal text), *Identifier* (identifiers are normalized to a common symbol), *Literal* (literals are normalized to a common symbol), and *Identifier + Literal* (identifiers and literals are normalized to a common symbol).
- **Selection:** Ranking and filtering of clones is used to cope with the usually large result sets. The (confounding) selection variable reflects this behavior with the following common clone selection approaches: *Random* (clones are selected randomly), *Lines* (clones are selected in ascending order according to the number of lines they span), *Blocks* (clones are selected in ascending order according to the number of blocks they include).

5.1.2. Response Variables. Each clone was evaluated, 1) whether it is relevant for refactoring and, 2) to which degree it associates to the four natures: Aspect, Structural, Syntactical or Logical. The resulting five response variables are given by a 5-point symmetric Likert scale ranging from *Strongly disagree* to *Strongly agree* with the neutral midpoint *Neither agree nor disagree*. This evaluation scheme is based on the findings of Walenstein et al. [38] that human raters do not agree on whether a clone should be refactored or not as different developers have different emphasizes. The Likert scale mitigates this issue by avoiding a binary

TABLE 2: Intraclass Correlation Coefficient (ICC) of the expert responses measured by a two-way model with a fixed set of k raters.

			95% Confidence Interval		F Test			
Response	Type	ICC	Lower Bound	Upper Bound	Value	df1	df2	Sig
Aspect	ICC3k	0.819	0.789	0.845	5.528	479	958	0
Logical	ICC3k	0.902	0.886	0.916	10.193	479	958	0
Structural	ICC3k	0.962	0.955	0.967	26.080	479	958	0
Syntactical	ICC3k	0.505	0.423	0.577	2.019	479	958	0
Relevance	ICC3k	0.800	0.767	0.829	4.999	479	958	0

Number of subjects = 480 Number of raters = 3
Two-way consistency averaged-measures ICC

decision and providing different levels of association and disassociation. Each response is evenly mapped onto a scale between -1 and 1 and averaged through all raters. This results in interval scale data with respect to the raters but also to the number of clones inspected within each group enabling the usage of standard statistical methods [39].

- **Aspect:** Clones of this nature contain statements related to cross-cutting concerns, e.g., debugging, logging, permission and authentication, data monitoring, etc. These clones are often unavoidable and cannot be removed with common clone refactoring strategies. Aspect Oriented Programming (AOP) frameworks are a solution to these clones and a general review on AOP methods is given by Kurdi [40], while Bengtsson [41] describes an approach specialized for IEC61131-3.
- **Logical:** Code fragments of logical nature describe an algorithmic unit fulfilling a specific task. They contain a dense occurrence of computations and operations on data structures nested within control flow constructs.
- **Structural:** Code fragments are of structural nature if they exhibit many definitions and initializations. They build up the structure of a software system. Typical examples are class, struct or variable definitions or initializations in header files or global constant definition files.
- **Syntactical:** Clones of syntactical nature are the result of text-based detectors that do not interpret whitespace or syntactical symbols (braces, brackets, etc.). For example, series of closing braces belonging to deeply nested control flow constructs may be detected as a clone by a text-based detector.
- **Relevance:** Relevance captures the likelihood that an expert would issue a refactoring of a particular clone in a general maintenance scenario. It reflects the typical true and false positive classification but avoids the forced binary decision.

5.2. Procedures

A subset of the found clones from Section 4 was selected and presented to three experts. Each expert was briefed in the meaning of the response variables. Each rater was free to evaluate the clones on his own pace and the inspection sessions were done self-managed. Each of the experts had a very strong background in software engineering. The average experience of the experts was 11.33 years (SD $= 4.04$ years).

Evaluation procedures computed the Inter-Rater Reliability (IRR) to quantify consistency and agreement among experts. Further, a linear model was fitted to expose relationships between the relevance of clones and the other variables. Finally, a set of hypothesis tests were conducted to give a further incentive on whether tool support specific for IEC 61131-3 languages are justified.

5.3. Evaluation

Overall 480 clones distributed over 32 groups (2 languages $\times$ 4 options $\times$ 4 selection approaches) with each containing 15 clones were inspected. This results in a total of 1440 inspections (480 clones $\times$ 3 raters) conducted by the experts. A two-way, consistency, averaged, Intraclass Correlation Coefficient (ICC) measure [42] was used to assess the reliability of the 1440 inspections with respect to the nature and relevance. Results within Table 2 show that there was a high degree of agreement among the expert ratings over the 480 clones. The consistency was excellent (Cicchetti interpretation guidelines [43]), except for the Syntactical nature only being fair ($ICC3k_{Syntactical} = 0.505$). Given the high ICC, a minimal amount of measurement error was introduced by the experts affecting the power of subsequent analysis. Ratings on the syntactical nature were deemed too erroneous therefore excluded from further analysis.

Figure 2 shows the expert averaged inspection result distributions for the natures *Aspect*, *Logical* and *Structural* as well as for *Relevance*. The left facet captures inspections of clones detected only via the *Text* mode of the detector. On the right facet are inspections of clones detected with additional normalizations (*Support*) in place, i.e., normalization of identifiers, literals or both. The distributions indicate that the experts had a clear idea whether a clone is positively associated with a nature or not. This can be seen by the slim bellies in the neutral region of the response scale.

Clones are not associated with the *Aspect* nature for C/C++ in the text mode. This slightly changes with the usage of normalization support indicated by the longer tail of the violin shape. ST clones are stronger associated with the *Aspect* nature indicted by the third quartile reaching beyond the neutral region of the response scale. However, the central tendency is still in the disagree range for both detector capability modes. *Logical* clones are scarce and

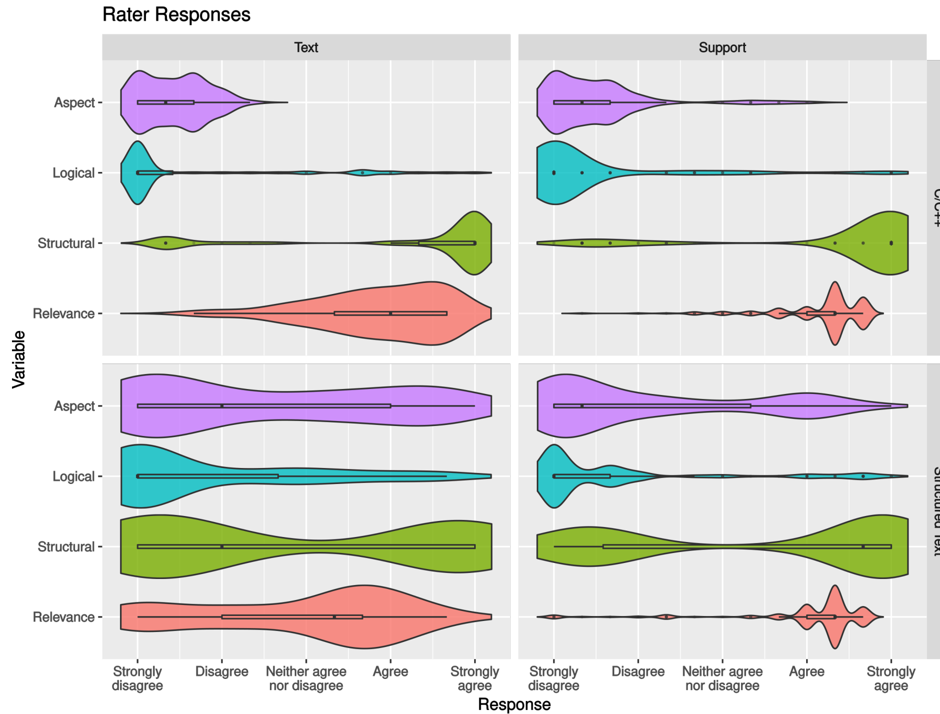

FIGURE 2: Responses with respect to the languages and natures. For easier interpretation the plot uses the Likert labels on the response axis (x-axis) although being values between -1 and 1. The Support facet includes responses from *Identifier, Literal* and *Identifier/Literal* option.

the central tendency for C/C++ and ST are both in the strongly disagree area. The outliers indicate the few clones that contain algorithmic content. Most *Logical* clones were found in the text mode for ST, nevertheless, the tendency is still towards a disassociation. Many clones found in C/C++ are of *Structural* nature indicated by the median located at strongly agree. The normalization support increases the number of *Structural* clones even more. For C/C++ the first quartile moves towards the strongly agree region, for ST a shift from dissociation to an association in the central tendency of the responses is measured.

Relevance is slightly worse for ST compared to C/C++ as the median shows, nevertheless using the additional support removes this offset and places both into the strong positive range.

5.3.1. Linear Model. A multivariate linear regression was calculated to predict *Relevance* based on *Language*, *Option*, *Selection* but also through the natures *Logical* and *Structural*. (Note: *Syntactical* has been excluded because of an insufficient reliability of the ratings and *Aspect* did not reach significance.) The maximum positive response of *Relevance* is 1 for a perfect association, 0 for neutral and -1 for a perfect disassociation. A significant regression equation was found ($F(28, 451) = 40.04, p < 2.2 \cdot 10^{-16}$), with an R^2 of $.713$. Interactions between *Language* and *Option*, *Option* and *Selection*, and between *Selection* and the included natures were significant. Regression residuals show acceptable departures from normality and parallel lines as a pattern. Patterns were expected because of the Likert scale being averaged by only three raters. An unacceptable variation in the variances was detected, therefore heteroscedasticity corrected hypothesis tests were conducted.

The linear model shows a strong positive logarithmic relationship to the number of lines a clone spans, increasing its relevance by 0.1 for each magnitude in lines. Clones from ST have a lower base relevance compared to C/C++ clones (-0.22) but strong positive significant interactions ($0.18 - 0.26$) with the different options. Similar, interactions that constitute blocks and normalizations in which identifiers are normalized (Identifier, Identifier+Literal), are positive significant with estimates between $0.29 - 0.37$. All these coefficients indicate combinations of options, selection methods and languages that greatly increase the relevance of clones. In terms of nature, there were strong significant coefficients that represent the interaction between *Structural* or *Logical* with the (between file) blocks selection method.

5.3.2. Statistical Tests. The planned tests investigated whether there is a statistically significant effect in *Relevance* between different groups of clones. Contrasts measure effects within and between languages given the text mode and the average of all normalizations but also the effects of selection methods. Table 3 contains contrasts and their respective hypothesis tests with *free* [44] adjusted p values that account for multiple comparisons. Test 2 and 3 compare the differences between text mode and additional support within the two languages where both reach significance, although ST with a larger effect. No marginal significant effect between random selection, and block or line oriented selection approaches could be found (Test 4). However, the low p-value and the existence of interactions indicate that significant effects between specific levels of the variables are present. Tests 6, 7 and 8 represent between language tests

TABLE 3: Significance tests of specific contrasts with respect to the depended variable Relevance.

	Contrast					95% Confidence Interval			
	Language	Option	Selection	Mean Diff.	Std. Error	Lower Bound	Upper Bound	Sig.	
1	Language	Text - Support	Selection	-0.225	0.034	-0.316	-0.135	$4.82 \cdot 10^{-10}$	***
2	C/C++	Text - Support	Selection	-0.116	0.035	-0.209	-0.023	0.005	**
3	ST	Text - Support	Selection	-0.335	0.058	-0.488	-0.182	$6.78 \cdot 10^{-8}$	***
4	Language	Option	Random - (Blocks, Lines)	0.063	0.027	-0.011	0.136	0.074	.
5	Language	Option	Lines - Blocks	0.007	0.051	-0.128	0.143	0.985	
6	C/C++ - ST	Option	Selection	0.052	0.031	-0.030	0.136	0.202	
7	C/C++ - ST	Text	Selection	0.217	0.067	0.040	0.394	0.005	**
8	C/C++ - ST	Support	Selection	-0.002	0.030	-0.083	0.079	0.986	

Sig. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1   Adjusted p values – free method

with no marginal effect (Test 6). Test 7 shows a significant difference of clones between the two languages given that they were detected with the text mode. Test 8 shows that this significance is not found if clones are detected with additional normalizations.

5.4. Interpretation

The results show that the tool support has a positive effect on the relevance of clones. This can be seen in Figure 2 where the median and first quartile moves into the strong positive region, but also in the hypothesis tests. This positive effect is given in the within language tests (Test 2 & 3) but also in between languages tests where the initial significant difference is removed by the additional normalization features. The between language effect is most likely caused by the header files (.h-files) of C/C++ that introduce more structural duplicates, which in hindsight are often relevant (linear model coefficient).

The selection approach does not influence the relevance of clones on average. However, there were positive effects associated with selection methods based on the number of blocks that are shared between files.

Most clones are of *Structural* nature and the usage of normalizations increases their total proportions making them more likely to be encountered. *Logical* clones are inversely proportional to structural clones and therefore less often encountered with normalizations. Clones of *Aspect* nature are mostly found with a low minimum line count of clones but remain strongly dependent on the application context. The nature of clones between the languages is fairly similar with mostly structural clones and some logical clones. ST code contained more aspect clones nevertheless these are less prevalent if normalizations are used.

6. Threats to Validity

The study faces threats to validity that might reduce the power of the analysis. First, the generalization of results is limited because only one software system was analyzed. However, the system is a real-world example and the applied development approach can be considered representative for many other evolving software systems for industrial automation [7]. Furthermore, the choice of the analysis tool and its implementation, as the experts and their specific background in software development may also have introduced a bias. Finally, the confounding configuration problem discussed by Wang [45] might be an issue. We chose the minimal line count for a clone fairly low such that aspect clones are easier spotted.

7. Summary and Conclusions

In this paper, we presented the results from the analysis of code clones in a real-world PLC software system, which has been evolved over several development iterations as part of a large industry project. The software system contained code written in C/C++ and in IEC 61131-3 Structured Text.

We found that clones do exist in PLC software systems regardless of the applied programming language. Awareness for clones is an important aspect of professional software development, independent whether they are viewed positive or negative. Industry projects require support for detecting, tracking and managing clones as software systems evolve. Similarly to previous studies [7], we can also conclude that the existing tool support for PLC languages with respect to clone detection is insufficient. Furthermore, we found that language adaptations for detectors, that enable the use of normalizations, improve the relevance of clones significantly. This is especially true for maintenance scenarios focusing on structural deficiencies. Concluding, companies that develop PLC systems can justify investments in clone detector adaptations. These investments widen the range of clone detection, analysis and management tools and strengthen professional software development within the industrial automation industry.

Future work includes methodologies for efficient filtering of clones based on their nature through complexity metrics and the repetition of the study on other PLC software systems including systems from different industry partners.

Acknowledgments

The research reported in this paper has been supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET Center SCCH (FFG #844597).

References

[1] S. Cass, "The 2016 Top Programming Languages," 2016. [Online]. Available: http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages

[2] C. K. Roy and J. R. Cordy, "A Survey on Software Clone Detection Research," *Queen's School of Computing TR*, vol. 115, p. 115, 2007.

[3] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Proceedings - International Conference on Software Engineering*, 2009, pp. 485–495.

[4] W. H. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray, *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[5] M. Fowler and K. Beck, *Refactoring : improving the design of existing code*, ser. The Addison-Wesley object technology series. Boston (Mass.), San Francisco (Calif.), Paris: Addison-Wesley, 1999.

[6] R. Koschke, "Frontiers of software clone management," in *Frontiers of Software Maintenance, 2008. FoSM 2008.* IEEE, 2008, pp. 119–128.

[7] B. Vogel-Heuser, A. Fay, I. Schaefer, and M. Tichy, "Evolution of software in automated production systems: Challenges and research directions," *Journal of Systems and Software*, vol. 110, pp. 54–84, 2015.

[8] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[9] S. Harris, "Simian - Similarity Analyser," 2003. [Online]. Available: http://www.harukizaemon.com/simian

[10] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pp. 368–377, 1998.

[11] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," 2013.

[12] R. Koschke, "Survey of Research on Software Clones."

[13] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, no. c, pp. 109–118, 1999.

[14] M. Rieger, S. Ducasse, and M. Lanza, "Insights into system-wide code duplication," in *Proceedings - Working Conference on Reverse Engineering, WCRE*, 2004, pp. 100–109.

[15] M. Asaduzzaman and C. Roy, "VisCad: flexible code clone analysis support for NiCad," *Proceeding of the 5th*, pp. 77–78, 2011.

[16] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, "Gemini: Maintenance support environment based on code clone analysis," *Proceedings - International Software Metrics Symposium*, vol. 2002-Janua, pp. 67–76, 2002.

[17] R. Tairas, J. Gray, and I. Baxter, "Visualization of clone detection results," *Proc. ETX at OOPSLA*, pp. 50–54, 2006.

[18] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "ARIES : Refactoring Support Tool for Code Clone," *3-WoSQ Proceedings of the third workshop on Software quality*, pp. 1–4, 2005.

[19] C. Kapser and M. W. Godfrey, "Aiding Comprehension of Cloning Through Categorization," *Proceedings of the Principles of Software Evolution, 7th International Workshop*, pp. 85–94, 2004.

[20] C. Kapser and M. W. Godfrey, "Improved tool support for the investigation of duplication in software," *IEEE International Conference on Software Maintenance, ICSM*, vol. 2005, pp. 305–314, 2005.

[21] F. Li, G. Bayrak, K. Kernschmidt, and B. Vogel-Heuser, "Specification of the requirements to support information technology-cycles in the machine and plant manufacturing industry," in *IFAC Proceedings Volumes (IFAC-PapersOnline)*, vol. 14, no. PART 1, 2012, pp. 1077–1082.

[22] E. Duala-Ekoko and M. P. Robillard, "CloneTracker: Tool Support for Code Clone Management."

[23] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Enhancing clone-and-own with systematic reuse for developing software variants," *Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME 2014*, pp. 391–400, 2014.

[24] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An Ethnographic Study of Copy and Paste Programming Practices in OOPL," *Empirical Software Engineering, 2004. ISESE '04. Proceedings. 2004 International Symposium on*, pp. 83–92, 2004.

[25] C. J. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful: Patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, pp. 645–692, 2008.

[26] B. Baker, "On finding duplication and near-duplication in large software systems," *Proceedings of 2nd Working Conference on Reverse Engineering*, pp. 86–95, 1995.

[27] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code."

[28] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein, "Pattern matching for clone and concept detection," *Automated Software Engineering*, vol. 3, no. 1-2, pp. 77–108, 1996.

[29] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a\nsoftware system using metrics," *Software Maintenance 1996, Proceedings., International Conference on*, pp. 244–253, 1996.

[30] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the harmfulness of cloning: A change based experiment," *Proceedings - ICSE 2007 Workshops: Fourth International Workshop on Mining Software Repositories, MSR 2007*, pp. 0–3, 2007.

[31] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto, "Software quality analysis by code clones in industrial legacy software," *Proceedings - International Software Metrics Symposium*, vol. 2002-Janua, pp. 87–94, 2002.

[32] J. Krinke, "Is cloned code more stable than non-cloned code?" *Proceedings - 8th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2008*, pp. 57–66, 2008.

[33] J. Krinke, "Is Cloned Code older than Non-Cloned Code?" 2011.

[34] F. Rahman, C. Bird, and P. Devanbu, "Clones: What is that smell?" *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 503–530, 2012.

[35] G. M. K. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, and Y. Zou, "Studying the impact of clones on software defects," *Proceedings - Working Conference on Reverse Engineering, WCRE*, pp. 13–21, 2010.

[36] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, "An empirical study on inconsistent changes to code clones at the release level," *Science of Computer Programming*, vol. 77, no. 6, pp. 760–776, 2012.

[37] N. Göde and J. Harder, "Clone stability," *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pp. 65–74, 2011.

[38] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia, "Problems creating task-relevant clone detection reference data," in *Proceedings - Working Conference on Reverse Engineering, WCRE*, vol. 2003-Janua, 2003, pp. 285–294.

[39] G. Norman, "Likert scales, levels of measurement and the "laws" of statistics," *Advances in Health Sciences Education*, vol. 15, no. 5, pp. 625–632, 2010.

[40] H. A. Kurdi, "Review on Aspect Oriented Programming," vol. 4, no. 9, pp. 22–27, 2013.

[41] K. Bengtsson, B. Lennartson, O. Ljungkrantz, and C. Yuan, "Developing control logic using aspect-oriented programming and sequence planning," *Control Engineering Practice*, vol. 21, no. 1, pp. 12–22, 2013.

[42] K. O. McGraw and S. P. Wong, "Forming inferences about some intraclass correlations coefficients," *Psychological Methods*, vol. 1, no. 1, pp. 30–46, 1996.

[43] D. V. Cicchetti, "Guidelines, criteria, and rules of thumb for evaluating normed and standardized assessment instruments in psychology," *Psychological Assessment*, vol. 6, no. 4, pp. 284–290, 1994.

[44] P. H. Westfall, R. D. Tobias, and R. D. Wolfinger, *Multiple comparisons and multiple tests using SAS*. SAS Institute, 2011.

[45] T. Wang, M. Harman, Y. Jia, and J. Krinke, "Searching for better configurations: a rigorous approach to clone evaluation," *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, p. 455, 2013.