State-of-the-Art Report on Type III Clone Refactoring Techniques

Muslim Chochlov

13 November, 2019

Abstract

Clone refactoring is a source code transformation activity that can be used to remove code duplicates while preserving original behaviour/-functionality. For example, refactoring of three features in our industrial partner's product family allowed us to preserve functionality, reduce the code footprint and improve the time domain analysis module's performance for some of their systems. But the effort expended in this activity also indicated the need for refactoring semi-/automation. In this report, one existing literature review and a subsequent literature search (to probe for new work) were used to identify the state-of-the-art in clone refactoring techniques. Fifteen such papers were found, characterized, and reviewed in detail, highlighting their advantages. Additionally, previous refactorings we employed at our industrial partner's were described and empirically evaluated.

The review of existing clone refactoring techniques suggests that they can semi-/automatically refactor certain clones. Yet, they cannot account for all clone differences and higher levels of automation can result in reduced quality of refactored code (through boilerplate/redundant code). Likewise, code refactoring of three features from our industrial partner's product family suggests that some refactorings can be automated, whilst preserving code quality, whereas the other, such as "super-functionality", can require complex rules/developer's knowledge. Hence, a semi-automation approach is proposed for discussion, which includes both a heuristic-based part for the refactorings that can be automated, while preserving code quality, and an artificial neural-network based model to learn refactorings for more complex code differences and their transformations.

Abbreviations

ANN ..... Artificial neural network

AP ..... Add parameter

AST ..... Abstract syntax tree

CAF ..... Common analysis framework

CC ..... Clone class

CDT ..... Clone detection technique

CFG ..... Control flow graph

CP ..... Clone pair

CR ..... Clone refactoring

CRT ..... Clone refactoring technique

CS ..... Code smell

EM ..... Extract method

FTM ..... Form template method

GS ..... Google Scholar

KLOC ..... Thousand(s) of LOC

LOC ..... Line(s) of code

OOP ..... Object oriented programming

PDG ..... Program dependence graph

PM ..... Parameterize method

PUM ..... Pull up method

RA ..... Refactoring activity

RS ..... Reorder statements

SF ..... Super functionality

1 Introduction

Refactoring[1] is a transformation of source code that preserves the original behaviour/functionality of that code [28]. The goal of refactoring is to improve the software quality as part of software maintenance and evolution and/or a software re-engineering process [22]. Certain refactoring activities (RA) can be distinguished as part of refactoring process such as identification, scheduling, application, and others [22]. In this report, the expressions "refactoring" and "refactoring application activity" are used interchangeably and carry the same meaning (actual transformation of source code, while preserving behaviour), unless explicitly stated otherwise.

The primary targets for refactoring are certain degraded structures [2, 4, 17, 27] in source code (that can indicate the possibility for refactoring), usually referred to as code smells (CS) [13]. Fowler et al. described 22 such classes of CSs [13]. A code clone (further "clone") is one of these CS classes. Here, clones are pieces of code, identical to each other up to a certain degree [16].

The software product family of our industrial partner was initially created in an clone-and-own manner[2] and, as a result, common functionality was duplicated across the software systems, creating feature clones. Currently, the adopted re-engineering strategy for this product family includes identification of common functionality (locating features in SystemA and their clones in SystemB and SystemC) and subsequent refactoring of these clones towards a common code base. Manual refactoring of three features has allowed us to significantly reduce the code footprint and to improve the module-related performance for some systems in the product family. It has also streamlined the company's maintenance process somewhat, where one piece of code can now be maintained instead of three. It has emerged, however, that such clone refactoring (CR) requires substantial effort and it was decided to consult the existing research literature for semi-/automation approaches to refactoring.

The goal of this report is to review the state-of-the-art in clone refactoring techniques (CRT) (semi-/automated approaches to CR) and to recommend those that could work for our industrial partner's product family based on previous refactoring work. Particularly, this report focuses on CRTs that can refactor Type III clones (clones that have lines of code added, removed, or modified beyond variable renaming and type/literal change), as it was found that this type of clones is prevalent in the product family [6]. For this purpose, one existing literature review was analyzed and an additional search and review of existing work in CR was conducted, to identify more recent, relevant literature.

This report is structured as follows: Section 2 describes the literature review process, and Section 3 discusses existing CRTs in detail. Section 4 revisits

[1] The term "refactoring" was originally used in the context of object oriented programming (OOP), whereas the term "restructuring" was used in the literature to define similar activities in code written using other non-OOP languages [1]: in this report both terms are used interchangeably.

[2] The development branch of SystemA was forked to produce the other two systems: SystemB and SystemC.

the refactoring that was performed in our industrial partner's product family, and Section 5 concludes this report with a refactoring proposal, for discussion amongst the project team.

2 Literature Review Process

The literature review process for this report was organized as follows:

- An existing literature review on CR by Mondal et al. [23] was consulted and papers describing Type III CRTs were selected, from that review, for this report.
- An additional literature search was conducted (looking for literature going to August 2019 inclusively) to ensure no important literature is missing that describes Type III CRTs.
- The papers selected from Mondal et al. [23] and from the literature search were joined, duplicates removed, and the final list of literature was obtained, by looking for other papers on Type III CRTs, using backward and forward references of those papers already selected.

2.1 Identification of CRTs from an Existing Literature Review

Although the field of refactoring appears to be widely studied, to the best of our knowledge there is only one existing literature review that focuses, in part, on CR application activity [23]. Mondal et al. have conducted a survey to review existing literature in CR and clone tracking. They have reviewed papers starting from 1998 and going to 2017 inclusively, searching for those papers in seven well-known digital databases[3]. The authors ultimately selected 77 papers that in their opinion are relevant to CR.

As part of their review, Mondal et al. have identified three papers that describe approaches suitable for Type III CR [23]. However, a review of the other literature they selected suggested that there are other (Type III CR relevant) approaches considered. Mondal et al. [23] have discussed these potentially relevant papers in the following sections of their paper: "Automatic Refactoring of Code Clones", "Integrating Clone Detection and Refactoring", and "Semi-automatic Refactoring of Code Clones". Therefore, the papers discussed by Mondal et al. [23] in these sections were reviewed and those that satisfy both of the criteria below were selected:

- Papers that explicitly discuss techniques or tools that facilitate the application of RA.

[3]IEEExplore, ACM Digital Library, ScienceDirect, SpringerLink, Wiley Online Library, World Scientific, and The IET

- Papers that explicitly mention applicability of their techniques or tools towards either Type III or "near-miss" clones[4].

Following the selection process above, seven papers were selected for review. Therefore, after this step 10 papers were identified in total: three as explicitly mentioned by Mondal et al. [23] and the other seven as a result of implicit mention by Mondal et al.

2.2 Identification of CRTs from a Literature Search

Mondal et al. covered 77 CR related papers in their review [23]. However, the degree to which it covered the CR literature is open to question because:

- The review does not seem to follow a systematic literature review approach [23].
- The review covers papers going to 2017 inclusively: newer, relevant research may now be available.

For these reasons, it was decided to conduct an additional systematic literature search and review to see if there is any potentially important work missing using the following procedure:

- Google Scholar[5] (GS) search engine was used to search for relevant literature. The search engine allows for simultaneously searching multiple digital libraries (including those analyzed by Mondal et al. [23]).
- The following search strings were used to search with GS: "clone refactoring", "duplicate code refactoring", "clone restructuring", and "duplicate code restructuring".
- The time period had an open start date and would include all papers going to August 2019 inclusively.
- It was decided to rank the returned results "by relevance" (GS does not specify explicitly how "relevance" is calculated).

Because the resultant list is very large (over several thousands of entries returned) the following inclusion criteria were used for selection process:

- The paper had to describe an approach, a technique, or a tool for CR and it had to be explicitly directed towards source code.
- The paper had to be published in a peer-reviewed venue.
- The paper had not been previously analyzed in Section 2.1.

[4]Near-miss clones can refer to Type II and/or Type III clones combined. Therefore, when selecting the papers it was checked if particular authors investigate Type III clones as part of near-miss clones.

[5]`https://scholar.google.com`

The selection process was stopped after hitting ten consecutive irrelevant (not satisfying inclusion criteria) papers. As a result, 21 papers were found and selection criteria used previously in Section 2.1 (focusing on application RA and Type III clones) were applied. As a result, two matching papers were selected for detailed review.

2.3 Using Already Identified Papers to Find Relevant Literature

The 12 previously identified papers (10 from Section 2.1 and 2 from Section 2.2) were used as starting points to further search for potentially relevant literature using backward and forward citation references. Particularly:

- For backward citations: "Related work" or similar sections of these 12 papers were used to identify potentially relevant work.
- For forward citations: GS "Cited by" functionality was used to find the relevant work that cited each of these 12 CR papers.

The selection criteria, discussed in Section 2.1 (application RA and Type III clones), were applied when choosing the relevant literature. This process was applied iteratively to all newly found papers until no more relevant literature was found. As a result, three additional papers were selected. Therefore, after following the procedures in Section 2.1, Section 2.2, and in this section, 15 relevant papers, in total, were selected for detailed review.

3 Detailed Review of Type III CRTs

3.1 Characterization of Type III CRTs

For better illustration and comparison, the CRTs discussed in the 15 reviewed papers were characterized using attributes adopted and/or adapted from existing clone detection/refactoring reviews [23, 29]. Also, to characterize evaluation of these CRTs, attributes from existing feature location reviews were adopted [7][6]. Additionally, three attributes (not encountered in the literature characterizing clone detection techniques (CDT) or CRTs) were added: "Type", "Refactorings", and "Results". The "Type" attribute was added, because during the review process, existence of two methodologically different groups of CRTs was found: recommender systems and heuristics-based semi-/automated CRTs. The "Refactorings" attribute was used to allow for better illustration of common code transformations supported by a technique. The "Results" attribute was used to highlight how a technique compares to other CRTs, the effectiveness of a technique in terms of its runtime performance, and the evidence of how CR performed using the technique: aspects important for this

[6] Although feature location approaches are different from CRTs, evaluation attributes used in that review seem to be generally applicable to CRTs' evaluation

Table 1: Refactorings frequently encountered in reviewed approaches

Refactoring	**Described in**	**Description**
Extract Method (EM)	[13]	Extract sets of (common) statements and place those in a separate method
Parameterize Method (PM)	[13]	Merge similar methods introducing a parameter(s) for varying values
Form Template Method (FTM)	[13]	Merge similar methods (usually creating a method in a superclass) moving varying parts to external methods (having the same signature and usually placed in subclasses).
Pull Up Method (PUM)	[13]	Remove similar methods from subclasses and merge these in a superclass
Add Parameter (AP)	[13]	Add parameter to a method: can be used to control execution of varying parts
Reorder Statements (RS)	[18]	Reorder statements in a block of code (usually to obtain a contiguous piece of common statements)

Table 2: Source code representations frequently encountered in reviewed approaches

Representation	**Description**
Abstract syntax tree (AST)	A tree that represents source code's syntactic structure. Some details, like end of line delimiters can be omitted.
Control flow graph (CFG)	A directed graph, where nodes are basic blocks (linear sequence of statements, for example) and where edges represent control flow between these blocks.
Program dependence graph (PDG)	A directed graph, where nodes are usually statements or expressions and where edges represent either data or control flow dependencies.

report. The full list of all attributes, their abbreviations, and their possible values is presented below:

- *Type (T)*: shows the type of a CRT. The value is a binary choice between "recommender" (R) CRTs and "semi-/automated" (SA) CRTs.
- *Refactorings*: shows the types of source code transformations that a technique can produce. Table 1 shows refactorings frequently encountered in reviewed approaches, references work where these refactorings were first described, and provides short descriptions. For refactoring values not listed in the table, "Other" value is used.
- *Supported language(s) (Sup.L)*: shows programming languages that are

supported by a technique. The values come from the set of existing programming languages.

- *Code representation (C.Rep)*: shows how source code is represented internally by a technique. Common representation values include an abstract syntax tree (AST), a control flow graph (CFG), or a program dependency graph (PDG) (see Table 2 for descriptions). For code representations not listed in the table the value "Other" is used.

- *Clone granularity (Cl.Gr)*: shows the clone granularity supported by a technique. The values come from a finite set: block (B), method (M), function (F), or class (C).

- *Clone set size (Cl.SS)*: shows the number of similar clones in a group that a technique can process. The value is binary: either a clone class (CC) or a clone pair (CP). A CC is a set of similar clones, whereas a CP is a special instance of a CC: a pair.

- *Tool/source availability (Avail)*: shows if a tool or the source code (from which the tool can be built) is available for a technique. Flag "Y" is used to indicate availability.

- *Subject systems*: shows software systems to which a CRT was applied and/or which were used to evaluate that CRT. It has four sub-attributes:

  - *Licensing (Lcs)*: shows the type of license. Three possible values are: proprietary (P), open source software (OSS), or "Mixed".
  - *Number of Systems (#)*: shows how many systems were used.
  - *Size*: shows the size of a system(s) in thousand(s) of LOC (KLOC). In case multiple systems were used, the size is shown as a range: the size of the smallest system followed by the size of the largest system.
  - *Number of Clones (#C)*: shows the number of clone groups (either CPs or CCs) extracted from subject systems to which a technique was applied and/or which were used for evaluation. The values are "S" for a number of clone groups with a range of [1, 100], "M" for clone groups from a range of [101, 1,000], "L" for clone groups from a range of [1,001, 100,000], and "L+" for numbers above 100,000.

- *Evaluation*: describes how a technique was applied/evaluated. It has six sub-attributes (for each of these flag "Y" indicates truth):

  - *Comparison (Cmp)*: shows if a technique was compared to other existing approaches.
  - *Academic evaluation (A)*: indicates if a technique was evaluated/applied by academic participants.
  - *Professional evaluation (P)*: indicates if a technique was evaluated/applied by industry professionals (e.g. developers).

  - *Quantitative evaluation (Qn)*: indicates if metrics/statistical data was used to evaluate and describe a technique.
  - *Qualitative evaluation (Ql)*: indicates if qualitative aspects were evaluated.
  - *Performance (P)*: explicitly indicates if a technique's performance was measured as part of quantitative evaluation.

- *Results*: describes the results of the evaluation/application of a technique. It has three sub-attributes:
  - *Improved effectiveness (E+)*: shows in percentage terms how much the CR effectiveness of a technique improves as compared to other technique in evaluation (the values are percentages).
  - *Code refactored (C.R)*: indicates (using a "Y" flag) if source code was refactored as part of the evaluation.
  - *Performance (Perf.)*: shows the performance of a technique. The values here are free form notes in which "min." stands for minutes and "sec." for seconds.

Because of high dimensionality, the characterization of CRTs is split between two tables. Table 3 contains attributes up to and including "Tool/source availability" in the order listed above and focuses on CRTs. Table 4 contains the remaining attributes in the above order and focuses on the evaluation of CRTs. In each table, papers were grouped by the CRT they describe: this improves readability and analysis, as some papers only differ in evaluation and some don't have substantial differences (for example, see "Narasimhan" in both tables: the papers describing this approach are grouped together). In both tables, the first "Technique" column refers to a CRT either by the name of a CRT (italic) or if there is no name then by the name of the first author (bold) of the paper where it was first described. The CRTs are ordered by the time they first appeared in the literature. For every technique, a set of papers is provided in the second column "Paper ref.": again these papers are ordered by the time they appear in the literature. For example, in Table 3, *JDeodorant* is the name of a technique described in four related papers [20, 21, 31, 32]: these are grouped together and ordered by the time of appearance. For any attributes for which values are not available,a blank field is used.

3.2 Recommender CRTs

Recommender CRTs (see Type (T) "R" in Table 3 and Table 4) can analyse structural relationships between clones and based on that can suggest an appropriate refactoring. For example, a recommender CRT can identify that two method clones belong to two subclasses (extended from a common superclass) and then suggest a PUM refactoring.

Koni-N'Sapu proposed SUPREMO, a technique that supports CR with the help of textual and visual aids [19]. SUPREMO defines a set of scenarios

Table 3: Characterization of Type III CRTs reviewed in this report

Technique	Paper ref.	T	Refactorings	Sup.L	C.Rep	Cl.Gr	Cl.SS	Avail
Fanta	[11]	SA	Other	C++	Other	F/C	CC	
SUPREMO	[19]	R	EM, PM, FTM, PUM, Other	C++, Ada, Java, Smalltalk	Other	B/M	CC	
LIME	[33]	SA	EM, PUM		AST	B/M	CC	
Gorg	[14]	SA	FTM		AST	M	CC	
SPAPE	[3]	SA	EM, PM, RS	C	AST/PDG	B	CP	
JDeodorant	[20]	SA	EM, PM, RS	Java	extended PDG	B/M	CP	Y
	[31]	SA	EM, PM, RS	Java	extended PDG	B/M	CP	Y
	[21]	SA	EM, PM, RS	Java	extended PDG	B/M	CP	Y
	[32]	SA	EM, PM, RS, FTM	Java	extended PDG	B/M	CP	Y
Narasimhan	[24–26]	SA	EM, PM, AP	C++	AST	M	CC	Y
DCRA	[12]	R	EM, FTM, Other	Java		B/M	CP	
Ettinger	[9,10]	SA	EM, RS	Java	CFG/PDG	B	CP	Y

Table 4: Evaluation of Type III CRTs reviewed in this report

Technique	Paper ref.	Subject systems				Evaluation						Results	
		Lcs	#	Size	#C	Cmp	A	P	Qn	Ql	Pr	E+	C.R	Perf.
Fanta	[11]	P	1	120	L			Y					Y	
SUPREMO	[19]	Mixed	9	2-300			Y		Y	Y			Y	
LIME	[33]													
Gorg	[14]													
SPAPE	[3]	OSS	10	17-67	M	Y	Y		Y		Y	110%		3 min. max
JDeodorant	[20]	OSS	7	50-120	M	Y	Y		Y			83%		
	[31]	OSS	9	50-210	L+		Y		Y		Y		Y	<1sec. for 98% cases
	[21] [32]	OSS	9	50-210	L		Y		Y				Y	
Narasimhan	[24–26]	OSS	10		S			Y		Y			Y	
DCRA	[12]	OSS	4		M		Y		Y					
Ettinger	[9,10]	OSS	4		M		Y							

where each scenario describes certain structural relationships between clones at method and block level granularity. For each such scenario a set of refactorings is suggested (where refactorings are those proposed by Fowler et al. [13]). For example, given a scenario "in the same class" where two methods within a

class contain duplicated code the following refactorings can be suggested: EM, "Insert Method Call", PM, and/or FTM[7]. The textual editor in SUPREMO allows for pairwise comparison of clones and displays additional statistics for these clones such as suggested refactoring(s), number of matching lines, density of these matching lines, and the number of similar clones within a given CC. Internally, SUPREMO relies on two existing tools: Moose[8] (static source code analysis tool and parser) and DUPLOC CDT [8]. Therefore, programming language support in SUPREMO is limited to those supported by Moose: at that time these were C++, Ada, Java, and Smalltalk. The tool was evaluated using nine software systems ranging from 2,000 to 300,000 LOC. Unsurprisingly, the author found that clones with higher density of matching lines and higher number of such lines are worth refactoring. Scenarios where method clones were located in a single class or in sibling classes seemed to prevail and accounted for over 50% of all scenarios. Additionally, a developer used SUPREMO to refactor three systems and decided to apply suggested refactorings to 71%, 37%, and 35% of detected clones for each system respectively.

Similarly to Koni-N'Sapu [19], Fontana et al. proposed a CR recommender technique, DCRA, that suggests scenario-based refactorings and CPs to be refactored [12]. This technique can be applied to block/method level CR in Java source code. It extends the work by Koni-N'Sapu by ranking the CP candidates for refactoring using their weights. The weights are determined by the gains in LOC from potential refactoring and by the compliance of refactoring to OOP principles. For example, a PUM refactoring that preserves inheritance and significantly reduces LOC would get a high score. DCRA was used with four open source systems (231 CPs total) and according to the authors was able to correctly (the authors have manually assessed correctness) recommend refactoring for 160 such CPs.

Recommender CRTs seem to be able to suggest correct refactorings (and in case with DCRA also to prioritize those refactorings) in some situations: although the evidence for this is still scarce. These approaches, however, do not seem to semi-/automatically refactor source code, instead focusing on guiding a developer. For actual application of refactoring a developer's involvement is still required.

3.3 Running Example: a Clone Class

To assist discussion of semi-/automated CRTs (Type "SA" in Table 3 and Table 4) a simple example of a CC is introduced in Figure 1. In the figure, a CC consists of three similar Python methods. Code pieces across a CC can be divided into "common" and "varying" parts: the former are exactly similar across a CC, whereas the latter have at least some variation. For example, in Figure 1 common code pieces are highlighted in green and the varying parts are left white.

[7]The author claims that SUPREMO can apply refactorings to source code for some cases, however, these cases are never explicitly mentioned.

[8]`http://moosetechnology.org/`

Listing 1: Clone 1

```
1 def some_method():
2     a = 0
3     b = 0
4     g = 0
5     c = 0
6     g += c
7     a += 1
8     b *= 2
```

Listing 2: Clone 2

```
1 def some_method2():
2     a = 0
3     b = 0
4     f = 0
5     c = 0
6     a += 1
7     b *= 2
```

Listing 3: Clone 3

```
1 def some_method3():
2     a = 0
3     b = 0
4     flag = True
5     c = 0
6     if flag:
7         flag = False
8     a += 1
9     b *= 2
```

Figure 1: Python clone class example

3.4 Semi-/Automated CRTs

Semi-/automated CRTs refactor clones according to pre-defined heuristics and usually support a set of code differences that can be refactored. As can be seen from Table 3, the majority of these techniques employ EM, PM or RS refactorings and can be applied to block or method level granularity CPs/CCs. For Type III clones, RS refactoring can be used to attempt moving varying parts outside of clone region (see lines 4, 6 in Clone 1, line 4 in Clone 2, and lines 4, 6-7 in Clone 3 from Figure 1) to obtain contiguous common blocks of code [3, 31]. The obtained contiguous common blocks can be further refactored using EM, for example. Another approach adds controls to the execution of varying parts based on a context: for instance, by introducing a control flag [26] or by encapsulating varying code in anonymous functions and passing these as method parameters [32].

In an early work on CR, Fanta and Rajlich [11] describe removal of function and class clones in an industrial setting. The authors refactored clones in a software tool called PET, that was developed by Ford Motor Company to allow for mechanical component design. The tool was implemented using the C++ language and consisted of 120,000 LOC at that time. The authors

proposed a semi-automated approach to CR: a combination of custom tools and developer interactions. The tools particularly would allow for the following refactoring automation: function insertion, function expulsion, function encapsulation, function/variable renaming, and argument re-ordering. When dealing with functions that share common code (Type III CCs can be an instance of this scenario), the authors propose to use "function encapsulation" to extract that common code as a function. Following that, a single function candidate is selected among those extracted functions (done manually by a developer) and the argument lists of the remaining functions in that CC are aligned to match that selected function (supported by a tool). Next, all calls in the code are replaced to match that selected singleton candidate and the clones are removed: both actions are automated via tool-support. From an implementation perspective, the tools developed by Fanta and Rajlich are supported by GEN++[9], a static C++ source code parsing and analysis tool (GEN++ allows for representation of source code as an abstract semantic graph). The authors report removal of clones in 10,000 LOC of PET (the exact number of refactored LOC is not reported). The tools that were used in this work do not seem to be publicly available.

Zibran and Roy presented an idea of LIME: an Eclipse plugin that allows for clone detection, refactoring, and management [33]. The plugin would parse the source code and represent it as an AST. Type I, II, and III clones are then detected and clustered into CCs. The authors propose to refactor Type III clones of block and method level granularity using two refactorings proposed by Fowler et al. [13]: EM and PUM. Because at the time of writing their paper it was ongoing work no evaluation is provided and there is no implementation available.

Gorg proposed a template-based idea to refactor method-level CCs [14]. In this approach, a CC is replaced with a template that generates clones at compile time (in essence code's footprint in LOC is being reduced, but the compiled code still contains clones). A template consists of common unchanged code (common for all clones within a CC) and placeholders that represent varying clones' parts. The approach consists of three artifacts that are extracted from Type III clones: a domain-specific model, meta-model, and generator. The model contains clone-specific variations and the meta-model specifies the structure of those variations. The generator is then used to substitute the placeholders with the actual clone-specific data. For example, varying ASTs can be specified in the model/meta-model to provide clone-specific behaviour. This approach, however, hasn't been implemented or evaluated.

Bian et al. proposed a semantics preserving approach to Type III CR [3]. The approach, SPAPE, first attempts to move varying code pieces outside a clone block (either before or after the block) to obtain contiguous common code pieces (RS refactoring). To achieve that, SPAPE transforms ASTs of a CP into two PDGs and moves varying parts if there are no data/control dependencies. If there are dependencies between common and varying code

[9] `https://web.cs.ucdavis.edu/~devanbu/genp/`

pieces, SPAPE attempts to remove those dependencies by following two code transformation rules: duplicating conditional block (leaving one duplicate for cloned code and creating one duplicate for varying code outside a clone block) or duplicating loops (using similar heuristics as used for conditional block). In cases when such transformations cannot be applied, the varying parts within a CP are handled by introducing conditional control variables: such a variable can control which branch is executed. For example, in Figure 1 in Clone 3, lines 6-7 can be safely moved (without changing behaviour or producing side effects) before line 5. In Clone 1, however, line 6 cannot be moved because it is data dependent on line 5. A CP transformed in such a manner is then extracted and merged in a newly created method: EM refactoring. The places in code that call this newly created method are updated with respective method calls and arguments. SPAPE was compared to an earlier semantics-preserving algorithm introduced by Komondoor and Horowitz [18]. Both were used to refactor clones in 10 open source systems (ranging from 17,000 to 67,000 LOC) and SPAPE was found 110% more effective (being able to refactor more clones) with running time not exceeding 3 minutes. Yet, it should be noted that only 192/454 CPs were extracted by SPAPE. Also, the complexity of these CPs and their type is unclear. SPAPE, while automating CR also seems to introduce boilerplate code: for example, if there are many varying parts within a CP, SPAPE will create as many additional control variables and/or code branches hindering maintenance and evolution of this code in the future.

The latter issue was addressed by Krishnan and Tsantalis who proposed a technique that attempts to minimize the number of differences within a CP while maximizing the number of common cloned statements within that pair [20]. To accomplish this, their technique represents a CP as two extended PDGs (additional types of edges and object state variables added) and inspects all matching solutions between these two graphs. The solution with the maximum number of matches (common code) and the minimum number of differences (varying parts) is selected as optimal. The technique then applies RS refactoring in a similar way to Bian et al [3]. It also allows the refactoring of such differences as varying constant values using the PM refactoring (the technique can parameterize seven differences, in total). The technique was compared to CeDAR (a Type II CRT) [30]: both were used to detect clones in seven open source systems. The authors report that their technique was able to detect 83% more refactorable clones (344/954) than CeDAR. The authors do not report the types of clones or the effects of refactoring (e.g. source code reduction). In their further work, Tsantalis et al. [31] enhanced this technique, primarily by changing the code representation and supporting more non-trivial code differences for refactoring. For example, Krishnan and Tsantalis [20] supported seven differences: in this work 16 are supported (e.g. *this.method = a* can be recognized as similar to *setMethod(a)*). Another major contribution of this work was a study in which 610/2,306 CPs from nine Java systems were correctly refactored. These CPs were covered by unit tests: this was used to ensure that the behaviour of refactored code remained unchanged. In the second part of their study, the authors assessed 1,150,967 CPs for refactorability: these were obtained using four differ-

ent CDTs from the same nine subject systems. They reported that less than 7% of Type III CPs are refactorable using their approach. Additionally, runtime performance of their approach was reported: in 98% of the CP cases, it took less than one second for their approach to run. This approach was later implemented as an Eclipse plugin, JDeodorant[10] [21]. Finally, in the most recent development of this approach Tsantalis et al. proposed to use lambda expressions (anonymous functions in Java 8) to address statement/block level differences in cloned methods [32]. In this case different lambda expressions are passed as parameters to a merged method and substitute varying statements. The same nine Java systems from their previous work [31] were used to collect clones. First, the authors used their approach to refactor 12,602 CPs (covered by unit tests) from "JFreeChart" that were assessed as refactorable. Further, they assessed 18,402 Type III CPs from nine subject systems and approximately 60% of these were reported as refactorable (but not actually refactored or tested).

Narasimhan et al. proposed a CRT that operates at method-level granularity and can be applied to a CC [24–26]. The technique represents methods within a CC as ASTs and calculates the largest common subtree from these ASTs. This largest common subtree becomes a common code of a newly merged refactored method (EM refactoring). The varying parts are refactored using PM and AP. For example, in Figure 1 lines 1, 4, 6 in Clone 1, lines 1, 4 in Clone 2, and lines 1, 4, 6, 7 in Clone 3 are different across that CC. The difference in line 1 can be trivially refactored by renaming the methods' names. The other differences can be refactored by introducing a parameter(s) (PM/AP) that takes three values. For example, a branch or switch statement can be created that depending on the value executes either $g = 0$, $f = 0$, or $flag = True$ in the resultant merged method (original line 4 in the clones, see Figure 1). For other differences, (e.g. varying values) PM can be used, similar to Tsanatlis et al. [31]. The authors evaluated their approach using 10 open source systems from Github[11]. In the first phase of their evaluation they have refactored eight CCs in five C++ systems and created pull requests for these changes: at the moment of writing their paper only one refactored CC was accepted. They have repeated the process in the second phase with 10 CCs across the other five systems: this time nine CCs were accepted, 10/18 total pull requests across two phases accepted. The tool, implementing this approach seems to be available as an Eclipse plugin.

Ettinger et al. [9,10] proposed an improved algorithm to that introduced by Komondoor and Horwitz [18]. Such an algorithm takes source code input as a PDG and attempts to move varying parts outside a clone (either before of after) so that semantics/behaviour is preserved. A limitation to that approach [18] is that common code pieces have to be identified manually (e.g. code highlighted in green in Figure 1). Ettinger et al. [9] proposed an algorithm that can automatically find the largest refactorable subset in a CP. The authors claim that their algorithm improves the efficiency of the overall approach, while maintaining correctness: they have applied their approach to 110 CPs and 59 of these

[10] `https://marketplace.eclipse.org/content/jdeodorant`
[11] `https://github.com`

seem to be refactorable by their approach with results comparable to Komondoor and Horwitz [18]. In later work Ettinger et al. [10] theoretically proved the efficiency and correctness of their algorithm. The implementation of their approach/algorithm is available as a source code.

The advantages of semi-/automated CRTs:

- They can automatically refactor source code as evident by existing application/evaluation of some CRTs (see Table 4 for such approaches).
- These techniques seem to be able to significantly reduce the refactoring time. For example, the performance evaluation of CRTs conducted by Bian et al. [3] and Tsantalis et al. [31] show three minutes maximum runtime per subject system for the former and less than a second in 98% of the CP cases for the latter.

There are several disadvantages of using these approaches:

- They can require complex rules to account for various differences in Type III clones: they currently cannot account for all situations. For example, Tsantalis et al. in two separate works assessed 7% [31] and 60% [32] of investigated Type III clones as refactorable (it should be noted that these weren't actually refactored).
- They can introduce boilerplate code reducing code quality: almost all reviewed CRTs use common refactorings such as PM, AP, FTM, and RS that introduce additional/duplicate branches or loops and add parameters to control the execution of varying parts of the code, based on a given context.
- They still need a developer's interaction. The pull requests submitted by Narasimhan et al. [26], though trivial, still required additional fixes (as evident, for example, from this[12] pull request's commit history).
- From technical perspective: the most common code representations, such as AST, CFG, and PDG (see Table 3), require a full parser.

4 Revisiting Refactoring of Our Industrial Partner's Code

The output of the CoRA feature location technique consists of a feature (in SystemA) and its two clones (in SystemB and SystemC). Every feature (and its clones), in turn, consists of a set of subprograms. Similar subprograms across this feature-set (the feature and its clones) form a CC of subprogram level granularity. The size of a CC can vary from two (a CP) to three depending on the number of systems that have that functionality. Also, there can be subprograms that don't have clones (but still have to be refactored): these are

[12] `https://github.com/oracle/node-oracledb/pull/28`

referred to as "singletons". In this section, first an example CC[13] from our industrial partner's product family is presented and then it is used to illustrate common (as empirically observed by the author of this report) refactorings and their usage statistics.

4.1 An Example of a CC: "Obfuscated_subprogram"

"Obfuscated_subprogram" (a subprogram variant in SystemA) is a CC from the *VR* feature. This feature has already been refactored, tested, and merged into production code. This CC was selected as an example because it allows for illustration of common refactorings that were applied during the CR process and these are discussed in the next section. It contains three subprogram clones: "obfuscated_subprogram" for SystemA revision *"be561caa"*, "obfuscated_subprogram" for SystemB revision *"7ce8d06"*, and "obfuscated_subprogram" for SystemC revision *"5b72b18"*[14]. This CC was consolidated in a subprogram also called "obfuscated_subprogram" in a Common Analysis Framework (CAF) revision *"282a118"*.

4.2 Common Refactorings

For every CC/CP and singleton in a feature-set (a feature and its clones) returned by CoRA the following refactoring steps were performed[15]:

1. *Refactor a CC/CP/singleton.* In this step, a subprogram(s) was refactored and merged (applies to a CC/CP only), if possible, to obtain a single representation.

2. *Update external caller code.* The calls across the product family were updated to use the refactored subprogram(s). As part of this, additional changes to a software system could include modifications of data structures, introduction of "stub" subprograms, and changes to a caller's code.

3. *Remove subprogram-clones.* The old CC/CP/singleton code was removed from the product family.

This report is focused on the common refactorings that were applied in the first step. These can be divided in two groups: those that were applied per subprogram (without looking at other clones in a CC/CP or if there are no clones as is the case with singletons - referred to as "subprogram-scoped" (SS)) and CRs discussed throughout this report (applied per CC/CP). Below commonly used SS refactorings and their examples from "obfuscated_subprogram" CC are given:

[13]There is no need for a separate singleton example as all the refactorings that can apply to singletons can also apply to CCs, but not vice versa.

[14]The actual subprograms' names are different in the code.

[15]Non source code related modifications such as changes to configuration files are omitted here. Also the testing phase is omitted.

- *Remove an unused (dummy) argument/local variable.* An argument can be removed if other variables are not data-dependent on it and if the value of this argument doesn't change (and this change is visible to a caller). Argument *"random_argument"* in SystemC "obfuscated_subprogram" lines 4, 38, 183 can be safely removed (it is never used/changed inside the subprogram). Similarly, a local variable can be removed if other variables are not data-dependent on it and if the value of this variable is not returned (in a function, for example).

- *Replace module/common members with arguments (RWA).* Module/common members (variables or subprograms) are replaced with extra arguments to a subprogram if these members are not visible to the CAF library. For example, array *"random_array"* from module *"random_module"* in SystemA "obfuscated_subprogram" line 4 is refactored into argument *"random_array"* in CAF "obfuscated_subprogram" lines 4 and 30.

Commonly used CRs and their examples from "obfuscated_subprogram" CC are provided below:

- *Rename identifier/reserved keyword.* This refactoring is a generalization of Fowler's "Rename Method" [13] and seems to be used in other CRTs [26, 31] for Type II CR (clones that have different identifiers or constants/literals/values). It refactors a CC/CP as follows: for varying identifier's names across a CC/CP one unique name is created/selected. For example, in SystemA "obfuscated_subprogram" line 145 "random_subprogram" is called, in SystemB "obfuscated_subprogram" line 134 "randomsubprogram" is called, and in SystemC "obfuscated_subprogram" line 166 "randomsubprogram" is called: this method call is renamed as "random_subprogram" in CAF "obfuscated_subprogram" line 183. A special instance of this type of refactoring is renaming of syntactically different, but semantically identical reserved keywords: for example, ".lt." is semantically equal to "<".

- *PM* (see Table 1). This refactoring can be used to address constant/literal differences in a CC/CP. For example, in SystemA "obfuscated_subprogram" line 79 "random_variable" is used in a logical expression, in SystemB "obfuscated_subprogram" line 72 "0" is used in that same expression, in SystemC "obfuscated_subprogram" line 86 "0" is used in the expression. Using PM this difference was refactored in CAF "obfuscated_subprogram" lines 4, 21, and 98 where an argument is introduced and later used in that expression.

- *AP* (see Table 1). This refactoring can be used standalone or as part of other refactorings such as PM or module/common member replacement. For example, in CAF "obfuscated_subprogram" lines 7, 51, and 200 *"some_other_subprogram"* argument is added (and later used).

- *Use super-functionality (SF).* This refactoring extends the functionality of a piece of code, if possible, adding and merging similar statements/blocks

Table 5: Common refactoring statistics across the 38 cases.

Refactoring	**# Cases**
SS/RWA	9
CR	4
CR/SF	2
CR/RWA/SF	22
CR/RWA/SF/EM	1

of code from other clones across a CC/CP (Type III clones). For example, in SystemC "obfuscated_subprogram" statements in lines 122-123 form a subset of statements from SystemA "obfuscated_subprogram" lines 115-119 and from SystemB "obfuscated_subprogram" lines 104-109. These can be merged (if functionality is preserved) as shown in CAF "obfuscated_subprogram" lines 137-141.

- *Optimization.* This refactoring refers to various optimizations performed. For example, lines 79-84 in SystemA "obfuscated_subprogram", lines 72-77 in SystemB "obfuscated_subprogram", and lines 86-91 in SystemC "obfuscated_subprogram" are removed from a loop and a return statement is added to allow for early termination, as can be seen in CAF "obfuscated_subprogram" lines 98-103.

4.3 Refactoring Statistics

For the three features previously refactored, 42 refactoring cases (including both singletons and CCs/CPs) were identified altogether. To four of these, refactoring was not applied: it was decided to leave three in the software system(s) code and one was dead code that was removed. For the remaining 38 cases (of which 9 are singletons, 27 are CCs of size 3, and 2 are CPs), common refactoring(s) are summarized in Table 5, using the following abbreviations:

- *SS*: indicates if only a subprogram-specific refactoring(s) was applied.
- *CR*: indicates if, in addition to SS, CR was applied.
- Additionally, it is indicated if RWA and SF refactorings (see previous Section 4.2) were used (because the author of this report has empirically observed these to require significant amount of refactoring effort). Also, it is shown, for comparison, if EM refactoring (see Table 1) was used as it is frequently described in CR literature (see Table 3).

As can be seen from the table, all the cases required refactoring: no singletons/CPs/CCs were moved to CAF as-is. RWA seems to be frequently used across all the cases: successfully applied in 32/38 cases. For CCs/CPs, SF refactoring was also frequently used: successfully applied in 25/29 cases. Only in one case, EM refactoring was additionally applied.

5 Conclusion: Proposing CRTs for Our Industrial Partner's Product Family

In this report existing Type III CRTs were reviewed with the goal of suggesting an approach for semi-/automation of CR across the product family. One existing literature review was used to identify CRTs proposed prior to and including 2017. An additional literature search was conducted to include newer work, going to August 2019. Also, to ensure no important literature is missing, backward/forward references of identified papers were inspected for relevant papers. Fifteen CRTs were selected for detailed review and two methodologically different types of CRTs were found: recommender approaches (see Section 3.2) and semi-/automated approaches (see Section 3.4). The latter are more relevant to our goal (semi-/automation) and their essential characteristics are summarized below:

- These approaches seem to be capable of effectively refactoring certain code differences in CCs and therefore can reduce refactoring effort.
- In these approaches, complex code differences seem to be addressed using higher levels of automation. This, in turn, seems to result in increased amount of boilerplate code and can decrease software quality.
- Complex heuristics can be required to account for all possible differences across CCs.
- A complete language parser is required to support these CRTs.

Empirical evaluation of refactoring activities (see Section 4 and Table 5) highlighted the following peculiarities:

- All 38 cases required refactoring.
- RWA and SF refactorings are both prevalent (RWA refactorings were applied to 32/38 identified cases (both singletons and CCs/CPs) and the vast majority (25/29) of CCs/CPs were merged using SF refactoring) and time and effort consuming (as observed by the author).

Based on the analysis of relevant CRTs and on the empirical evaluation of refactoring activities, a semi-automated CRT combining heuristics-based refactorings and an artificial neural network (ANN) based learning can be employed. This technique would allow for automation of some SS refactorings such as RWA, removal of unused variables, and CRs such as renaming, PM/AP: for these code-quality preserving refactoring heuristics can be created (possibly partially reusing existing work by Tsantalis et al. [32] and Narasimhan et al. [26]). The automation of more complex refactorings such as SF and optimization can require either complex (and likely infeasible for some cases, as refactorings are based on individualized human knowledge) heuristics or can produce large amounts of boilerplate code if naively automated. Instead, the proposed approach would

learn from existing refactorings using ANNs: a similar approach for program translation by Chen et al. [5] showed promising results. It could be pre-trained on existing accomplished refactorings and would continue to re-train as refactorings are performed, suggesting possible solutions to a developer. Recurrent neural networks that can translate sequence-to-sequence can be particularly suitable for this task [5]. There are two major caveats to this approach:

- For a heuristics-based part: a full parser is required to construct the source code's representation. However, because only a subset of source code has to be parsed (subprograms and partially modules), a subset of syntactic rules would have to be implemented.
- For an ANN-based part: small datasets can impact accuracy of ANNs. This can be remedied by applying transfer learning, for example: re-using a pre-built similar ANN model. Such an approach was successfully used by Google for multi-lingual ANN-assisted natural language translation [15]. One source of training data could come from Tsantalis et al. [32] who provide several thousands of CPs and their consolidated views.

References

[1] Mesfin Abebe and Cheol Jung Yoo. Trends, opportunities and challenges of software refactoring: A systematic literature review. *International Journal of Software Engineering and its Applications*, 8(6):299–318, 2014.

[2] Nour Ali, Sean Baker, Ross O'Crowley, Sebastian Herold, and Jim Buckley. Architecture consistency: State of the practice, challenges and requirements. *Empirical Software Engineering*, 23(1):224–258, 2018.

[3] Yixin Bian, Gunes Koru, Xiaohong Su, and Peijun Ma. SPAPE: A semantic-preserving amorphous procedure extraction method for near-miss clones. *Journal of Systems and Software*, 86(8):2077–2093, aug 2013.

[4] Jim Buckley, Sean Mooney, Jacek Rosik, and Nour Ali. Jittac: a just-in-time tool for architectural consistency. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1291–1294. IEEE, 2013.

[5] Xinyun Chen, U C Berkeley, and Dawn Song. Tree-to-tree Neural Networks for Program Translation. In *Advances in Neural Information Processing Systems*, pages 2547—-2557, 2018.

[6] Muslim Chochlov, Michael English, Jim Buckley, Daniel Ilie, and Maria Scanlon. Identifying Feature Clones in a Suite of Systems. In *Source Code Analysis and Manipulation (SCAM), 2018 IEEE 18th International Working Conference on*, pages 149–154, 2018.

[7] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code : a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[8] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 109–118, 1999.

[9] Ran Ettinger and Shmuel Tyszberowicz. Duplication for the removal of duplication. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016*, pages 53–59, 2016.

[10] Ran Ettinger, Shmuel Tyszberowicz, and Shay Menaia. Efficient method extraction for automatic elimination of type-3 clones. In *SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 327–337, 2017.

[11] Richard Fanta and Václav Rajlich. Removing Clones from the Code. *Journal of Software Maintenance and Evolution*, 11(4):223–243, 1999.

[12] Francesca Arcelli Fontana, Marco Zanoni, and Francesco Zanoni. A duplicated code refactoring advisor. In *International Conference on Agile Software Development*, volume 212, pages 3–14, 2015.

[13] Martin Fowler, Kent Beck, John Brant, and William Opdyke. *Refactoring Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[14] Torsten Görg. A Model-Based Approach to Type-3 Clone Elimination. *Softwaretechnik-Trends*, 32(2):33–34, 2012.

[15] Melvin Johnson, Mike Schuster, Quoc V Le, Maxim Krikun, Yonghui Wu, Zhifeng Chen, Nikhil Thorat, Fernanda Viégas, Martin Wattenberg, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's Multilingual Neural Machine Translation System: Enabling Zero-Shot Translation. *Transactions of the Association for Computational Linguistics*, 5:339–351, 2017.

[16] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. IEEE Trans Softw Eng. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[17] Tara Kelly and Jim Buckley. A context-aware analysis scheme for bloom's taxonomy. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 275–284. IEEE, 2006.

[18] Raghavan Komondoor and Susan Horwitz. Effective, automatic procedure extraction. In *11th IEEE International Workshop on Program Comprehension, 2003*, pages 33–42, 2003.

[19] Georges Golomingi Koni-N'sapu. *A Scenario Based Approach for Refactoring Duplicated Code in Object Oriented Systems*. PhD thesis, University of Bern, 2001.

[20] Giri Panamoottil Krishnan and Nikolaos Tsantalis. Unification and refactoring of clones. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014 - Proceedings*, pages 104–113, 2014.

[21] Davood Mazinanian, Nikolaos Tsantalis, Raphael Stein, and Zackary Valenta. JDeodorant: Clone refactoring. In *Proceedings - International Conference on Software Engineering*, pages 613–616, 2016.

[22] Tom Mens and Tom Tourwe. A Survey of Software Refactoring. *Software Engineering, IEEE Transactions*, pages 1–14, 2004.

[23] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. A Survey on Clone Refactoring and Tracking. *Journal of Systems and Software*, September 2019.

[24] Krishna Narasimhan. Clone merge - An eclipse plugin to abstract near-clone C++ methods. In *Proceedings - 2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pages 819–823, 2015.

[25] Krishna Narasimhan and Christoph Reichenbach. Copy and Paste Redeemed. In *Proceedings - 2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pages 630–640, 2015.

[26] Krishna Narasimhan, Christoph Reichenbach, and Julia Lawall. Cleaning up copy–paste clones with interactive merging. *Automated Software Engineering*, 25(3):627–673, 2018.

[27] Michael P O'Brien, Jim Buckley, and Christopher Exton. Empirically studying software practitioners-bridging the gap between theory and practice. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 433–442. IEEE, 2005.

[28] William Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992.

[29] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013.

[30] Robert Tairas and Jeff Gray. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Information and Software Technology*, 54(12):1297–1307, 2012.

[31] Nikolaos Tsantalis, Davood Mazinanian, and Giri Panamoottil Krishnan. Assessing the Refactorability of Software Clones. *IEEE Transactions on Software Engineering*, 41(11):1055–1090, 2015.

[32] Nikolaos Tsantalis, Davood Mazinanian, and Shahriar Rostami. Clone Refactoring with Lambda Expressions. In *Proceedings of the 39th International Conference on Software Engineering*, pages 60–70, 2017.

[33] Minhaz F Zibran and Chanchal K Roy. Towards Flexible Code Clone Detection , Management , and Refactoring in IDE. In *Proceedings of the 5th International Workshop on Software Clones*, pages 75–76, 2011.