State-of-the-Art Report on Clone Detection

Muslim Chochlov

December 4, 2017

Abstract

Code clones occur in source code, due to such coding practices as copy&paste and code re-use. Subsequently, code clones can impede software maintenance activities and so, to identify clones, a number of clone detection techniques were proposed. In this report, three existing literature reviews and a subsequent clone detection literature search in ACM and IEEE Xplore digital libraries were used to identify the pool of existing techniques. From these techniques 13 were selected (based on high citation index), assigned to seven classes, and reviewed in detail. Additionally, the advantages and limitations for each class of techniques were outlined.

The analysis of clone detection techniques indicates that there is still work that has to be done when improving detection of Type III and Type IV clones. Also, tree-based and graph-based techniques can require additional parsing of source code and the algorithms used in these approaches can have inappropriately large time complexity. Based on these conclusions, text-based (potentially augmented with structural information), information retrieval based, clone detection techniques are recommended to locate clones in our industrial partner's software systems: SystemA, SystemB, and SystemC.

Abbreviations

AST ..... Abstract syntax tree

CD ..... Clone detection

CDT ..... Clone detection technique

HT ..... Hybrid clone detection technique

IR ..... Information retrieval

KLOC ..... Thousand lines of code

LOC ..... Line of code

LSI ..... Latent semantic indexing

MBT ..... Metrics based clone detection technique

MLOC ..... Million lines of code

MOBT ..... Model based clone detection technique

PDG ..... Program dependence graph

PDGBT ..... Program dependence graph based technique

TKBT ..... Token based clone detection technique

TRBT ..... Tree based clone detection technique

TXBT ..... Text based clone detection technique

UML ..... Unified modelling language

1 Introduction

A code clone (further in the text referred to as "clone") is a piece of code, that is similar to another piece of code to a certain degree [11]. This coupling (between the clone and the other similar piece of code) is called a "clone pair". Clones can appear in a software system (or across several software systems in a product family) as a result of code re-use or as a result of using clone-prone programming approaches. Common forms of code re-use are [25]:

- Copy&paste;
- Re-use of functionality;
- Branching of the entire software system[1].

Programming approaches that can add clones to source code include [25]:

- Generative programming;
- Merging of software systems;
- Delay in restructuring.

As a a result of these practices, software systems can consist of a sizeable number of clones. For example, Rattan et al. [23] suggest that 20%-30% of code in large software systems is cloned. This negatively affects software maintenance [3, 13, 21]: modifications or bug fixes to an original piece of code must also propagate to its clones and often it is hard to find the location of those clones [23]. To reduce the number of clones, first they have to be identified/detected. In this report, clone detection is defined as identifying clones in the source code of a software system or across the software systems in a product family.

In large software systems, manual clone detection is infeasible [2]. Therefore, techniques that automate clone detection to a certain degree are needed. A clone detection technique (CDT) is a semi-/automated approach to clone detection.

Plenty of CDTs have been proposed in the literature [2, 23, 25]. The goal of this document is to report on the state-of-the-art in CDTs. (And the ultimate goal is to suggest the appropriate CDT(s) to be used in the second phase (CD) of our industrial partner's source code reunification project of its three systems: SystemA, SystemB, and SystemC.) To accomplish the goal of this report, three well-known literature reviews of CDTs are discussed [2, 23, 25] and subsequently this discussion was expanded to incorporate to newer work on CD.

This document is structured as follows: in Section 2, the literature review process is described. In Section 3, different classes of CDTs are reviewed in detail and their benefits and limitations are highlighted. Section 4 concludes this review.

[1] The latter is the case with our industrial partner's products: the development branch of SystemA source code was forked two times to produce SystemB and SystemC products respectively.

2 Literature Review Process

2.1 Identification of CDTs that Appeared before and Including 2011

Because there is a plentiful body of knowledge in the CD area built up over a considerable period of time, it is reasonable to rely on existing, well-known literature reviews for identification of existing CDTs. Three well-known literature reviews were selected and their details are shown in Table 1.

Table 1: The comparison of CD reviews.

	Bellon2007	Roy2007	Rattan2013
Credibility			
# Citations	588	503	141
Venue	Journal: Transactions on Software Engineering	N/A (technical report)	Journal: Information and Software Technology
Peer-reviewed	Yes	No	Yes
Venue H-index	137	N/A	76
Systematic literature review	No	No	Yes
Literature review scope and statistics			
# CDTs	19	31	73
Period of CD research covered	At most 2007	At most 2007	Up to and including 2011
Clone types described	3 types	4 types	9 types
Classification/taxonomy of CDTs	6 classes of CDTs	6 classes of CDTs	7 classes of CDTs

When selecting these existing reviews their credibility (via their citation index) was considered and their year of publication. While Bellon2007 and Roy2007 are dated, Rattan2013 updates these reviews with more recent CDTs and has a very large citation index for an article published four years ago. The scope of covered CDT papers was also considered and this was particularly impressive in the more recent Rattan2013 review (there is a broad representation of CDTs).

As can be seen from Table 1, all these reviews have high numbers of citations[2], which indicates their popularity and high relevance among CD research community[3]. Two of these reviews (by Bellon et al. and by Rattan et al. [2,23])

[2]These numbers were recorded by Google Scholar (`https://scholar.google.com/`) as of September 2017.

[3]It should be mentioned that sometimes a high citations number could be an indicative of

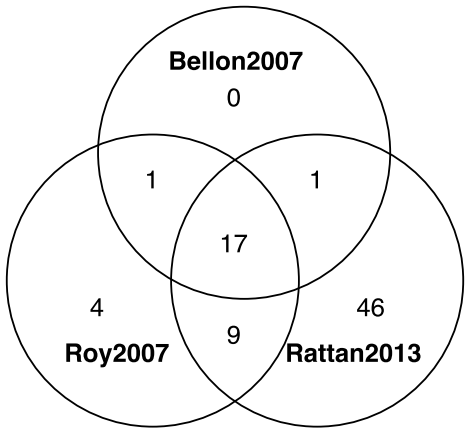

Figure 1: Intersection of CDTs reviewed by Bellon2007, Roy2007, and Rattan2013

.

were published in high quality (as indicated by their high H-indexes[4]), peer-reviewed venues. Additionally, the paper by Rattan et al. [23] is a systematic literature review paper. A systematic literature review is a type of literature review, originating in medicine research, that incorporates a rigorous methodology to answer research questions of that review [8, 14, 22]. Because of the rigorous methodology, this type of literature review has a number of advantages, such as reduced bias when selecting papers [22], and, hence, has been suggested for researchers and practitioners in software engineering [8, 14, 22].

Together these three literature reviews cover 78 unique CDTs (for a complete list of these CDTs see Appendix A: Listing and Classification of All CDTs Reviewed by Bellon2007, Roy2007, and Rattan2013). There is an overlap in the CDTs, covered by these literature reviews, as can be seen in Figure 1. The majority (73) of CDTs were reviewed by Rattan et al. [23], but there are also four CDTs that were reviewed by Bellon et al. only and one together by Bellon et al. and Roy et al. [2, 25].

In this report, no additional literature search was conducted to identify CDT papers, appearing in the literature prior to 2012. The 78 CDTs, identified by the three reviews discussed above, were considered exclusively when reviewing CDTs prior to 2012 because:

- The systematic literature review process was employed by Rattan et al. [23] (the peer-reviewed paper that reviews the majority of CDTs (73)). The authors explicitly used six top-quality, relevant, digital libraries to conduct their search and employed explicit search terms and relevant in-

a negative feedback (for example, a publication was heavily criticized).

[4]H-index in this case is the journal's number of articles (h) that received at least h citations. These indices were provided by Scimago (`http://www.scimagojr.com/`) as of September 2017

clusion criteria to select the papers. Such an approach objectively identifies high quality CDT papers.

- The remaining five CDTs come from two highly cited reviews [2, 25] and one of these reviews was peer-reviewed [2].
- The high overlap of CDTs identified by these three independent reviews (see Figure 1) suggests that there is high agreement on the CDTs that should be included pre-2007, a suggestion that is buttressed by looking at the year-of-publication of these CDTs. The majority of the remaining non-overlapping CDTs, referred to in Rattan et al. were post 2007 [23].
- In terms of the period covered, these three reviews cover CD research up to 2012.

The above reasons do not imply that all the relevant CDTs were identified by these reviews for the given period. However, the rigorous literature search process, the high quality digital libraries selected, the number of CDTs identified, and the fact that these reviews cover highly similar ground in terms of CDTs identified for a given period, suggests that the corpus of identified CDTs is a decent representation of CD research for that period. It also suggests that further analysis would only incrementally affect the outcome for that period.

2.2 Identification of CDTs that Appeared after 2011

The five years that have passed since the reviews suggest that the list of CD literature needs to be updated to include newer CDTs. To give a preliminary estimate of the volume of CDT papers that were published between 2012 and 2017 (both years inclusive), the following procedure was used:

- Two well-known digital libraries were selected: ACM[5] and IEEE Xplore[6]. These libraries were selected, because they were previously used in the CD review by Rattan et al. [23] and also because these libraries together store a large corpus of computer science related literature (arguably, the largest corpus).
- The search string "clone detection" was used with the search engines of these two digital libraries, mentioned above, to retrieve the documents that contain both search string terms ("clone" and "detection" in any order) in their titles.
- The search engines were instrumented to return only the papers that were published after 2011.

Using the procedure above, ACM search engine returned 51 matching documents and IEEE Xplore search engine returned 126 matching documents.

[5] `https://dl.acm.org`
[6] `http://ieeexplore.ieee.org/Xplore/home.jsp`

2.3 Classification of CDTs in This Report

Because of a large volume of CDTs, it is convenient to describe them using taxonomies. In taxonomies, individual objects are assigned to a class according to a set of characteristics common to this class. In CD, such characteristics are the type of source code representation and/or the type of clone matching employed by a CDT [2, 23, 25].

Table 2: Classes and the number of CDTs in these classes as used by Bellon2007, Roy2007, and Rattan2013.

Bellon2007	Roy2007	Rattan2013	This report
Text-based (2)	Text-based (5)	Text-based (11)	Text-based (11)
Token-based (4)	Token-based (4)	Token-based (10)	Token-based (11)
Tree-based (2)	Tree-based (5)	Tree-based (18)	Tree-based (17)
PDG-based (2)	PDG-based (3)	PDG-based (5)	PDG-based (6)
Metric-based (5)	Metrics-based (8)	Metrics-based (13)	Metrics-based (15)
Other (4)	Hybrid (6)	Hybrid (10)	Hybrid (12)
		Model-based (6)	Model-based (6)

The existing literature reviews, as discussed in Section 2.1, use slightly different names of classes (see Table 2) and do not always agree on the class of a CDT (for example, see the "Sim" CDT in Appendix A: Listing and Classification of All CDTs Reviewed by Bellon2007, Roy2007, and Rattan2013). For consistency, in this report the CDTs from these reviews were re-classified using the following rules:

- A "majority vote" was used when assigning a CDT to a class. For example, the "Sim" CDT was classified as token-based by Bellon et al. and Roy et al. [2, 25] and as tree-based by Rattan et al. [23] (see Appendix A: Listing and Classification of All CDTs Reviewed by Bellon2007, Roy2007, and Rattan2013). Because of the majority vote, Sim was classified as token-based in this report.

- If no majority vote could be obtained, the classification as per the more recent review was used. For example, the "clones/cscope" CDT was classified as hybrid by Roy et al. [2, 25] and as token-based by Rattan et al. [23] (see Appendix A: Listing and Classification of All CDTs Reviewed by Bellon2007, Roy2007, and Rattan2013). Because the review by the latter authors is more recent, clones/cscope was classified as token-based in this report.

As a result of the re-classification process, seven classes of CDTs, as shown in the final column of Table 2, are used in this report. The CDTs that appeared

after 2011 (see Section 2.2) were successfully assigned to one of these classes. If the authors of these CDTs explicitly specify the class then that class was used. Otherwise, a CDT was assigned to a class following the descriptions of the class as per Rattan et al. [23].

2.4 Selecting CDTs for Detailed Review

As follows from Section 2.1 and Section 2.2, the number of CDTs is large. Thus, it is not feasible to review all the relevant literature due to time constraints. Hence, it was decided to review in detail only a sub-set of these CDTs.

To select from the pool of 78 CDTs that appeared prior to 2012, the following selection procedure was used:

1. From each class of CDTs (see Table 2), select one[7] CDT that appears in the majority of reviews [2, 23, 25].
2. If there are several such CDTs (or if none of them appears in more than one review), pick the one with the highest citation count.

To select from CDTs identified in Section 2.2, the following selection procedure was used:

1. The documents returned by the ACM and the IEEE Xplore were sorted by the number of times they were cited.
2. From each of these two document lists, the five top-cited relevant papers were selected. The relevance of a paper was identified by the author of this report by reading the abstract and the introduction of the paper, if needed. The author applied the following criteria when assessing the relevance of a paper:
   - The paper had to present a novel/significantly modified CDT.
   - The paper had to provide the evaluation of the presented CDT.
   - The paper was not a follow-up paper to one of the already selected papers.
3. From the resultant set of 10 documents (five from the ACM results and five from the IEEE Xplore results) the final set of top five documents was selected for detailed review. If a paper occurred in both lists, it was automatically included in the final set. The remaining papers were included according to their citation count.

By following these procedures, eight pre-2012 CDTs [1, 5, 6, 11, 16–19] and five post-2012 CDTs were selected for review [9, 12, 20, 26, 27]. The selection procedures above ensured influential paper selection and also ensured every class of CDTs is represented by at least one technique.

[7] One exception to this rule was the selection of two textual CDTs [6, 18]. Though these two CDTs belong to the same class, their heuristics are substantially different. Thus, it was decided to include them both for a more thorough review.

2.5 Clone Types

CDTs are usually capable of identifying certain clone types. Clone types usually define how close (similar) two or more pieces of code are to each other [2, 25]. Rattan et al. additionally seem to use clone types to refer to a granularity of clones [23]. The complete list of clone types, as used by the three literature reviews (see Section 2.1) is shown in Table 3.

Table 3: Clone types as used by Bellon2007, Roy2007, and Rattan2013.

Bellon2007	Roy2007	Rattan2013
Type I	Type I	Type I
Type II	Type II	Type II
Type III	Type III	Type III
	Type IV	Type IV
		Structural
		Model
		Function

The following definitions of clone types are used:

- Type I: two pieces of code that are identical, bar white-spaces, comments, and layout.
- Type II: two pieces of code that, in addition to Type I differences, have also the names of identifiers and literals changed.
- Type III: two pieces of code that, in addition to Type II differences, have also LOC added, modified, and/or deleted.
- Type IV (semantic clones): two pieces of code that implement identical functionality, but otherwise are textually different.
- Structural clones reflect design similarities usually expressed at the level of software architecture [23].
- Function clones are limited to sub-program level granularity.
- Model clones are similar pieces in a model-based software systems.

In this report the first four types (Type I - IV) are used when describing CDTs, because they seem to align with degree of detection-difficulty.

2.6 Evaluation of CDTs

There are no standard methodologies when evaluating CDTs [23, 25]. Most often, to assess the effectiveness of a CDT, precision and recall are calculated [23, 25]. One way to calculate precision and recall for a CDT involves clone candidates. A clone candidate is a piece of code, returned by a CDT, that is a

potential clone. Then, the recall and the precision for a CDT is calculated, as shown in Formula 1 and in Formula 2 for recall and precision respectively. In these formulas, CC is a set of clone candidates and C is a set of known clones in a software system, a "gold set".

$$Recall = \frac{CC \cap C}{C} \tag{1}$$

$$Precision = \frac{CC \cap C}{CC} \tag{2}$$

Other evaluation criteria of CDTs, mentioned in the literature [25], involve:

- Portability: how applicable is a CDT across different programming languages and dialects.
- Scalability: how effective is a CDT in large software systems.
- Robustness: how effective is a CDT in identifying various clone types.

3 Clone Detection Techniques

3.1 Text Based Techniques

Text based CDTs (TXBT) analyze source code as textual data and can apply text pre-processing such as comments removal, white-spaces removal, and normalization [25]. In TXBTs, code pieces are usually compared to each other according to the similarity of their textual representation [23]. Usually, the comparison is done line-by-line, where strings belonging to different pieces of code are compared to each other, until some condition is met (e.g. a minimum number of similar lines is reached [25]). Textually similar (the similarity definition/threshold used by authors of TXBTs can vary) pieces of code are marked as clones and are returned as clone pairs.

A good example of a TXBT is the "duploc" approach, proposed by Ducasse et al. [6]. Duploc takes two source code files X and Y as an input and produces a report of clone pairs that were found in these files and visualizes the cloned code in the form of a scatter plot (in this plot, horizontal data represents LOC in X and vertical data represents LOC in Y). In the first step, duploc applies minimal transformations to lines in a file: comments and white-spaces are removed. In the second step, every line is compared to every other line for textual similarity: if a line is exactly similar to the other line then the result is "true", otherwise the result is "false". (Because comparing every line to every other line is computationally expensive (O(NxM) complexity in general case), the authors proposed calculating a hash for every line. The hash function produces an input-unique fixed size data value: all exactly similar lines will have the same hash value. These lines that have the same hash value are placed into a "bucket". Then, instead of line-by-line comparison, the hash value of a line could be matched to the common hash value of the bucket.) The results are

stored in a matrix NxM, where N is the number of LOC in file X and M is the number of LOC in file Y. The coordinates of the result in the matrix are $[n, k]$, where n is a line position in file X and k is a line position in file Y. For example, if a line 15 in the file X is compared to a line 10 in the file Y, then the result of this comparison is stored in the matrix cell with the coordinates [15,10]. In the resultant matrix, sequences of true values in the diagonals (top left to bottom right) indicate cloned code (a clone pair). Some of these sequences may have gaps ("false" values as a result of added/modified/deleted lines). Thus, when extracting clone pairs, the authors allow for the gaps of a certain size to occur in the sequences. The authors evaluated their technique using four software systems, all of which were written in different programming languages. They found that the average percentage of duplications in a file could range between 5.9% and 25.4% for different systems. The accuracy of these numbers is questionable though, because no "gold set" of known clones was used to evaluate the technique. Additionally, the technique demonstrated low performance in terms of running time: in a few cases the running time took up to seven hours to complete. The independent comparison study, conducted by Bellon et al. [2], showed that duploc was able to detect Type I and Type III secretly injected clones (duploc does not classify clones by type), showed average recall of 50% with a "cook" system, but failed to work on larger software systems.

A different TXBT was proposed by Marcus and Maletic [18]. Contrary to the duploc technique, the authors of this approach leverage comments (and also identifiers) that are available in source code. The input to their technique is the source code of a software system. It is partitioned into source code components (e.g. sub-programs). Each such source code component is then represented by a textual document. These resultant textual documents are organized into clusters according to their textual similarity. The information retrieval (IR) technique latent semantic indexing (LSI) is used to compare the documents. The authors do not compare their approach to other CDTs and to the best of our knowledge there are no such comparisons available in other studies. Roy et al. suggest that this technique is able to detect Type III and Type IV clones [24].

The major advantages of TXBTs in general are:

- Programming language independence: because these approaches are usually language-agnostic, they could be applied to a variety of source code files written in different programming languages.
- TXBTs usually require very little or no parsing of source code.

The limitations of these techniques include:

- Over-reliance on abundant and meaningful textual data: if such data is scarce/meaningless these techniques are less effective.
- Some TXBTs (e.g. duploc) are unable to detect clones that are very different textually (e.g. Type IV clones).

3.2 Token Based Techniques

Token based techniques (TKBT) transform the source code into a sequence of tokens. A token is a pair that has a name and a value to it. For example, in Table 4 every column (except the first column) represents a token. The upper row represents tokens' names and the bottom row represents tokens' values. A special program, called a tokenizer or lexer, is used to transform the source code into a sequence of tokens. Tokenizers take the source code as an input and follow the rules of the programming language to transform the source code into a sequence of tokens. If there are two or more similar sub-sequences (sub-strings) of tokens in the resultant sequence (string) of tokens, then these sub-sequences are clones. Because naive sub-string search is inefficient, TKBTs usually employ more effective string search data structures (and corresponding algorithms), such as a suffix tree [25]. A suffix tree stores all suffixes of a string S of length M, resulting in the tree having M leaves. Using a suffix tree, one can find a sub-string of a length K in the string S in O(K) time.

Table 4: An example of a Java statement tokenization.

Sequence of tokens	Identifier	Operator	Identifier	Operator	Literal	Separator	Comment
Sequence of source code characters (Java language)	x	=	a	+	2	;	// assignment

Kamiya et al. proposed a TKBT, CCFinder, that extracts clones from source code written in C/C++, Java, COBOL, and other languages [11]. The input to CCFinder are source code files. First, CCFinder transforms the source code files into a sequence of tokens: the output of the first step is a concatenated sequence of all tokens in a software system (every source code file is transformed into a sequence of tokens and these sequences are concatenated). In the second step the token sequence is transformed: certain tokens are added/removed/changed according to the transformation rules, specified by the authors. Some of these rules are:

- Removal of name-space related tokens.
- Addition of separator tokens to standardize the code. For example, according to compound block rule [11], in C-like languages, block separator tokens "{" and "}" will be added to one-line if-blocks, if missing.

In the following step, clone pairs are detected using the suffix tree data structure and sub-string search algorithms used with this data structure. Code pieces in a clone pair contain line numbers, that allow for identification of these pieces in source code files. The authors did not compare CCFinder to other CDTs. They reported on the performance of the technique: the processor time and memory usage increase linearly with the size (in terms of LOC) of a software system. In contrast, Burd and Bailey in their evaluation study of CDTs reported precision of 72% and recall of 72% for CCFinder [4]. Ducasse et al., however, reported more modest results with precision at 42% and recall at 43% [7].

More recently, another TKBT, Boreas, was proposed by Yuan and Guo [27]. First, Boreas filters irrelevant data from source code: string literals, comments, and include statements are removed. In the second step, tokens that represent variables, programming language reserved keywords, and symbols (e.g. separator, operator) are extracted from a piece of code. In the next step, metrics are calculated for each token representing a variable in a piece of code. For example, the number of variable's assignments, usages and occurrences in conditional/loop blocks is counted. The resultant metrics are stored in a "count vector" so that each variable has its own such vector. The set of these count vectors is stored in a "count matrix". In the fourth step, count vectors for reserved keywords and symbols are populated: the number of occurrences for each keyword and symbol is counted. To assess if two pieces of code form a clone pair, their cosine similarity is calculated using their count matrices of variables, their count vectors of reserved keywords, and their count vectors of symbols. Thus, similar code pieces can be organized into clusters. Boreas was used to detect clones in Java SE Development Kit 7 and Linux Kernel 2.6.38.6. The results were compared to another CDT, Deckard [10]. The evaluation showed that Boreas is able to identify a similar amount of cloned code (in terms of LOC) with precision comparable to that of Deckard. However, the authors of Boreas claim that it is more scalable: it requires less processor time and disk space.

Murakami et al. proposed a TKBT, CDSW, that identifies "gapped" clones (Type III clones) [20]. The input to the technique is a source code. First, the source code is transformed into a sequence of tokens. In the next step, hash values are calculated for every statement. (The authors define a statement as a sub-sequence of tokens that is separated by ";", "{", and "}" symbols.) Thus, the source code is represented by sequences of hash values. These sequences are compared using the Smith-Waterman algorithm. The algorithm identifies similar regions between two given sequences. The sequences that are similar above a given threshold form a clone pair. CDSW was evaluated using eight software systems and was compared to the eight other CDTs. The authors found that the recall and precision of CDSW were modest when compared to other techniques. However, the harmonic mean (a measure that evaluates precision and recall together) of CDSW was better than that of the other CDTs. This suggests that CDSW is a balanced CDT (in terms of precision and recall). The execution time for all the systems was below 30 seconds.

SourcererCC is a TKBT, proposed by Sajnani et al. [26], that particularly targets Type III clones in large software systems/systems' repositories. The technique represents the source code of a software system as a set of "code blocks". The authors define each code block as a piece of code within braces. The code block is then represented by a "bag-of-tokens" (similarly to a "bag-of-words" in IR: the order of tokens is not important, but their multiplicity is important). The similarity between two code blocks is calculated as the intersection between their bags-of-tokens. If the number of similar tokens is above some specified threshold, then the two code blocks are considered similar. Such an approach, however, is inefficient (especially when applied to large software

systems), because every code block has to be compared to every other code block in a system, resulting in $O(N^2)$ time complexity. Thus, the authors propose two optimizations:

- Tokens, representing a code block, are sorted in such a way that less frequent (more rare) tokens appear at the beginning of the token list. Then only sub-lists, containing the i first elements, are compared.
- Code blocks can be rejected as clone candidates without comparing all the tokens: the sum of current matches and the minimum number of unseen tokens in code blocks can be used. For example, if given two code blocks $B_1 = a, b, c$ and $B_2 = b, c, d, e, f$ and a threshold equal 3, then $1 + min(1, 4) < 3$ (token b is one match and there is one unseen token in B_1 and four unseen tokens in B_2) and hence these code blocks are not clones.

SourcererCC was compared to four other state-of-the-art CDTs. For scalability, the IJaDataset, containing 250 MLOC was used. It was found that SourcererCC scales well to large software systems, in some cases outperforming the other four CDTs. The recall of SourcererCC was assessed using the Mutation Framework and the BigCloneBench benchmarks [26]. With the Mutation Framework, SourcererCC showed 100% recall when detecting Type I, Type II, and Type III clones. With the BigCloneBench, SourcererCC showed high recall when detecting Type I and Type II clones, but much weaker recall when detecting Type III and Type IV clones. The authors also manually calculated the precision of SourcererCC and found that it is comparable with the other four CDTs.

The advantages of TKBTs are as follows:

- TKBTs can detect Type II clones: the source code can be abstracted using tokens and clones with renamed parts can be found.
- Execution times seem to be low when compared to other CDTs.

The limitations of TKBTs are:

- Tokenization is required to convert the source code in to a sequence of tokens. To support multiple programming languages, a tokenizer has to be created for each such language.
- In general, TKBTs do not seem to detect Type III and Type IV clones very accurately.
- Optimizations might be needed when using TKBTs in large software systems.

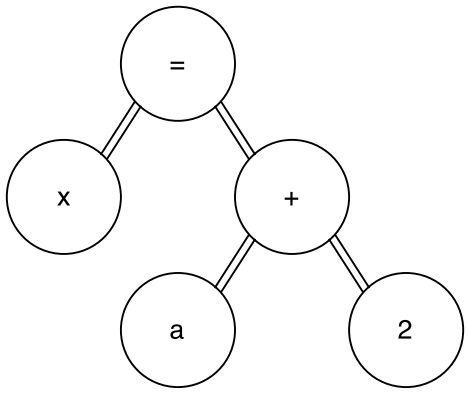

Figure 2: AST of the source code from Table 4

.

3.3 Tree Based Techniques

Tree based techniques (TRBT) represent source code as a full parse tree or as an abstract syntax tree (AST)[8]. A special program called a "parser" transforms source code into these tree-like structures according to the grammar rules of the programming language. For example, the AST of the piece of source code shown in Table 4 can look as shown in Figure 2. Once the source code is transformed in to a tree, similar sub-trees (clones) can be located.

Baxter et al. were one of the first to introduce a TRBT, CloneDR [1]. In the first step, the approach transforms the source code into an AST. Finding clones in this tree using naive sub-tree comparison is computationally expensive (the authors claim $O(N^4)$ time complexity). Thus, they proposed to categorize sub-trees with hash values and put them into B number of buckets: only the sub-trees in the same bucket need to be compared. According to the authors, such an approach allows for reduction of the time complexity down to $O(N)$. This approach, however, works for exact sub-trees and fails to detect sub-trees that have additional modifications (Type III clones). This happens because a good hash function will put such different sub-trees into different buckets. To solve this problem the authors proposed to use "bad" hash function, that ignores small trees in a sub-tree when calculating the hash value.

The authors haven't compared their technique to other existing CDTs. However, CloneDR was used in the comparison study by Burd and Bailey [4]. In that study, when compared to the other four CDTs, CloneDR demonstrated the highest precision of 100% and the lowest recall of 9%. The subsequent study by Bellon et al. [2] also showed that CloneDR finds only a small amount of clones (hence, small recall). Additionally, CloneDR showed poor scalability results (in terms of processor time and memory requirements) [2].

[8]The difference between the two is in the amount of details: in ASTs some details such as white-spaces and parentheses can be omitted.

More recently, another TRBT was proposed by Ishihara et al. [9] that allows for detection of method-level granularity clones. In the first step of their approach, the ASTs are created for every source code file in a software system and those sub-trees that correspond to methods are extracted. In the second step, these extracted sub-trees are normalized: variables and literals are replaced with special tokens to eliminate non-essential differences in the sub-trees. Further, methods that have no inner blocks (conditional/loop pieces of code) are excluded. The reason for exclusion is to remove smaller getter/setter[9] methods whose presence can introduce many false positive clone candidates. The remaining methods' ASTs are transformed into textual representations and hash values are calculated for each of them. In the final step, the methods are grouped according to their hash values: the groups of two or more methods indicates that they are clones.

The authors conducted a pilot study where the scalability of this technique was evaluated. The results show that the technique was able to complete in four hours against the 360 MLOC.

The advantages of TRBTs are:

- TRBTs can detect Type III clones.

The limitations of TRBTs are:

- Parser (and the corresponding grammar file) is needed to build an AST or full parse tree.
- Because of the previous point, TRBTs are not easily scalable for multiple programming languages.
- Execution times can be long, if a TRBT is not optimized [23].

3.4 Program Dependency Graph Based Techniques

Program Dependency Graph Based Techniques (PDGBT) represent source code as a program dependency graph (PDG). A PDG contains information about the data flow or the control flow in a piece of code. For example, a PDG can allow for tracing of a variable's assignment and usage in a piece of code: vertices of a PDG can represent statements where a variable is assigned/used and edges can connect these statements. PDGs are different from ASTs: the former represent semantic information about a piece of code, whereas the latter are built using the syntactic rules of a programming language [25].

Krinke proposed a PDGBT, "duplix", that represents the source code as a PDG and detects clones as similar sub-graphs in the PDG [16]. Vertices of the PDG are derived from those of an AST, except for the definitions of variables and procedures: these are assigned to special vertices. Also, the vertices in the PDG have additional attributes such as class, kind, and value. Additionally,

[9]Methods in Java language that set or return an object property value. These methods are generally small and pervasive.

control and data flow edges are added (e.g. vertices that assign/use/access a variable are connected). Then the problem of clone detection can be solved by finding isomorphic sub-graphs in such a PDG[10]. The problem of finding isomorphic graphs is NP-complete, thus the author proposed an approximation algorithm. The algorithm starts at two distinct vertices and inspects outgoing edges of these vertices. The edges that have the same attribute (class) are used to reach the next set of vertices. These vertices are added to a set and marked as the next vertices for inspection. The algorithm repeats until there are no outgoing edges of similar class or k number of steps has been reached. The resultant two sets of vertices belong to similar (isomorphic) sub-graphs. The author has evaluated duplix for scalability using 15 small (ranging from 2KLOC to 25 KLOC) software systems and has found that the execution times rise exponentially when increasing k values. In some cases, it took a significantly long time for duplix to complete (e.g. about seven hours to complete CD in a "twmc" software system that has 24,950 LOC). The study by Bellon et al. showed that duplix is able to detect Type III clones, but also reported poor scalability in terms of processing time [2].

The advantage of PDGBTs is similar to those of TRBTs: they can detect Type III clones.

The limitations of PDGBTs are:

- To construct a PDG source code has to be parsed: a parser is needed.
- PDGBTs are not easily scalable for multiple programming languages.
- Execution times can grow exponentially rendering PDGBTs inefficient with larger software systems.

3.5 Metrics Based Techniques

In metrics based techniques (MBT), characteristics (usually quantitative expressed as metrics) of clones are compared. As a precursor to that comparison, the source code is transformed into one of the representations discussed in the previous sections: text, tokens, ASTs, or PDGs. Various metrics for those representations are collected and accumulated into metrics' vectors. These vectors are then compared and similar vectors mean that a potential clone pair is found.

Mayrand et al. proposed a MBT, CLAN, to detect clones of function-level (sub-program-level) granularity [19]. The technique transforms the source code of each function in a software system into an AST and an AST is subsequently transformed into an "intermediate representation language". The latter form is used to collect the metrics that assess the name of a function, the layout of a function, the expressions of a function, and the control flow of a function. For example, the layout metrics show the amount of comments and blank lines in a function. The expression metrics can show the amount of external calls to other sub-programs and the number of declaration statements. The control

[10]In this context, the two graphs are isomorphic if there is a bijective mapping between their edges, attributes, and connected vertices

flow metrics can show the number of decisions and loops. For each function the values of these metrics are aggregated into a numeric vector. These numeric vectors are compared to detect similar functions (clones): if the vectors are exactly similar or if the delta of difference is below some specified threshold then the functions, represented by these vectors, are likely clones. CLAN was evaluated in two other studies by Bellon et al. and by Koschke et al. [2, 15]. Both studies reported that CLAN identifies a small amount of clone candidates (has a low recall). Also, Bellon et al. reported a high precision for CLAN [2] and found that CLAN was the most efficient when compared to the other five CDTs they tested (it never required more than four seconds to detect clones).

Characteristic vectors in MBTs are usually applied for the sub-tree matching and are used to detect Type III clones [23]. Aside from that the advantages and limitations of the underlying source code representation apply. For example, if ASTs are used to represent source code then the advantages and limitations of TRBTs will apply.

3.6 Model Based Techniques

Model based techniques (MOBT) represent source code using higher level concepts. For example, source code can be represented using unified modelling language (UML) or the Matlab's Simulink model[11]. Because usually these models have graph-like representation, graph-based approaches can be applied for CD in these models (see Section 3.4).

Deissenboeck et al. proposed a MOBT, CloneDetective/ConQAT, to detect clones in large systems that are designed using Simulink [5]. In Simulink, a system's model consists of blocks (which usually correspond to sub-programs) and interconnecting lines between these blocks. These lines specify data flow between the blocks (sub-programs). In large models, there can be repetitive similar clusters of blocks (similar blocks and interconnecting lines). Thus, CloneDetective attempts to find these "cloned" block clusters. In the first step, CloneDetective transforms a Simulink software system's model into a graph. Every vertex in this graph corresponds to a block and additionally stores block-type information. Every edge in this graph corresponds to an interconnecting line and also stores connection type information. Then the problem of CD in such a graph boils down to finding isomorphic sub-graphs, similarly to Krinke's "duplix" approach (see Section 3.4). As stated earlier, such a problem is NP-complete, thus Deissenboeck et al. use a breadth-first search approximation algorithm to locate isomorphic sub-graphs (similar to "duplix", see Section 3.4) [5]. The authors evaluated CloneDetective on a system containing approximately 4700 blocks and manually inspected the resultant clones. They claimed that the resultant clones are relevant for "practical purposes".

The advantage of MOBTs is that they can operate at a higher level of abstraction: identify cloned software components/sub-systems. However, similarly

[11] `https://fr.mathworks.com/products/simulink.html`

to PDGBTs these techniques can suffer from long execution times. Also, transformation of source code into an abstract form (e.g. UML) may be required.

3.7 Hybrid Techniques

Hybrid CDTs (HT) can combine the type of source code representation and/or the type of clone matching employed by the CDT types mentioned in the previous sections.

Leitao proposed a sub-program level HT, R2D2, to detect clones in software systems written in Lisp language (though the author claims that R2D2 can be extended to support other languages) [17]. R2D2 leverages TRBT and MBT to support CD. The technique transforms the source code into an AST and uses a series of specialized evaluations to compare two sub-programs: analysing identical forms, similar call sub-graphs, commutative operators, user-defined equivalences and transformations into canonical syntactic forms [17]. After each evaluation, evidence is obtained that two sub-programs are clones. This evidence is accumulated and the final likelihood of two sub-programs being a clone pair is estimated. The author applied R2D2 to SNePS legacy software system of 65KLOC written in Lisp. The author reported quadratic running time and claimed that the technique identified relevant clones (these were inspected manually by the author).

More recently Keivanloo et al. proposed a HT, SeByte, to identify Type III and Type IV sub-program level clones in Java bytecode [12]. SeByte is a combination of a TKBT and a MBT. The input to the technique is a Java bytecode file (compiled class file). In the first step, this bytecode file is transformed into a token sequence, but only certain tokens are retained (Java type names and called method names). The similarity between two sub-programs (represented as two token sequences) is calculated using two types of analysis: pattern matching and Jaccard coefficient. The first compares how similar are the two sequences of tokens preserving the ordering of tokens in a sequence. The second (Jaccard coefficient) calculates the ratio of token intersection between two sequences to their union (the ordering is not important in this case). The resultant clone pairs have to be above the specified similarity threshold. The authors used four software systems to evaluate SeByte. They also compared SeByte to the four other existing CDTs. The results indicated that the agreement rate between SeByte and the other approaches was 70% in the best case and 30% in the worst case.

HTs inherit advantages and limitations of CDTs that they combine. Thus, for example, if HT is a combination of TKBT and MBT the advantages and limitations of these approaches also apply to a HT.

4 Conclusions

In this report existing CDTs were reviewed with the ultimate goal of suggesting a CDT(s) to be used as part of code re-unification in our industrial partner's

software systems: SystemA, SystemB, and SystemC. Three existing literature reviews were used to identify CDTs proposed prior to 2012. Subsequently, ACM and IEEE Xplore digital libraries were used to identify CDT papers published after 2011. Of the pool of these CDTs, 13 were selected for detailed review. These 13 CDTs were assigned to an appropriate CDT class (see Section 2.3) and were reviewed within that class. The advantages and limitations for each class of CDTs were outlined. Summarizing the most essential points, it seems that:

- Detection of Type III and Type IV clones can still be a challenge for CDTs and especially for some TXBTs and TKBTs.
- TRBTs, PDGBTs, and MOBTs require additional parsing/analysis of source code that can be expensive.
- Time complexity of the algorithms employed by TRBTs, PDGBTs, and MOBTs can be inadequate, in that they take too long as the body of code scales up, and this can render these classes of CDTs inefficient for large code bases.

The prior analysis of our industrial partner's software systems (SystemA, SystemB, SystemC)[12] identified the characteristics of these systems that can influence the recommendation for a CDT. Here these characteristics are repeated:

- Type III clones are the most prevalent across the three software systems (SystemA, SystemB, and SystemC).
- The successful textual feature location in these systems suggested that plentiful, meaningful source code comments exist (and to a lesser extent identifiers).
- There are many source code irregularities (Fortran dialects, for example) and a full parser for such code is not readily available and can be costly to produce.

Based on these factors, an IR-based TXBT, similar to the TXBT proposed by Marcus and Maletic [18], is recommended for CD in our industrial partner's software systems. Such an approach leverages textual data in source code and has the following advantages:

- It can detect Type III and Type IV[13] clones.
- It also requires very little or no parsing.

[12]This analysis was carried out prior to this report and presented to the team working on code re-unification

[13]Given two different implementations, similar code comments can suggest semantic similarity

- The execution time is generally fast: the IR engine stores documents in an index for fast retrieval.

One caveat to this approach is that there has been no evaluation of this approach. However, its similarity to the feature location approach, that was used in the first report and that has demonstrated high precision and recall, can suggest that similar results can be achieved with the technique proposed in this report.

Finally, this approach can be hybridised (similarly to the feature location technique in the first report) to include structural information for CD.

References

[1] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377, 1998.

[2] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on software engineering*, 33(9):577–591, 2007.

[3] 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.

[4] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *SCAM 2002 - Proceedings, 2nd IEEE International Workshop on Source Code Analysis and Manipulation*, pages 36–41, 2002.

[5] Florian Deissenboeck, Benjamin Hummel, Elmar Jürgens, Bernhard Schätz, Stefan Wagner, Jean-François Girard, and Stefan Teuchert. Clone detection in automotive model-based development. In *Proceedings of the 13th international conference on Software engineering - ICSE '08*, pages 603—-612, 2008.

[6] 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.

[7] Stephane Ducasse, Oscar Nierstrasz, and Matthias Rieger. On the effectiveness of clone detection by string matching. *Journal of Software Maintenance and Evolution*, 18(1):37–58, 2006.

[8] T Dyba, BA Kitchenham, and M Jorgensen. Evidence-based software engineering for practitioners. *Software, IEEE*, 22(1):58–65, 2005.

[9] Tomoya Ishihara, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. Inter-project functional clone detection toward building libraries - An empirical study on 13,000 projects. In *Proceedings - Working Conference on Reverse Engineering, WCRE*, pages 387–391, 2012.

[10] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stéphane Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings - International Conference on Software Engineering*, pages 96–105, 2007.

[11] 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.

[12] Iman Keivanloo, Chanchai K. Roy, and Juergen Rilling. Java bytecode clone detection via relaxation on code fingerprint and Semantic Web reasoning. In *2012 6th International Workshop on Software Clones, IWSC 2012 - Proceedings*, pages 36–42, 2012.

[13] 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.

[14] Barbara Kitchenham, Emilia Mendes, and GH Travassos. A systematic review of cross-vs. within-company cost estimation studies. In *Proceedings of the 10th international conference on Evaluation and Assessment in Software Engineering*, pages 81–90, 2006.

[15] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone Detection Using Abstract Syntax Suffix Trees. In *2006 13th Working Conference on Reverse Engineering*, pages 253–262, 2006.

[16] Jens Krinke. Identifying Similar Code with Program Dependence Graphs. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 301—-309, 2001.

[17] António Menezes Leitão. Detection of redundant code using R2D2. *Software Quality Journal*, 12(4):361–382, 2004.

[18] A. Marcus and J.I. Maletic. Identification of high-level concept clones in source code. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 107–114, 2001.

[19] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of International Conference on Software Maintenance ICSM-96*, pages 244–253, 1996.

[20] Hiroaki Murakami, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. Gapped code clone detection with lightweight source code analysis. In *IEEE International Conference on Program Comprehension*, pages 93–102, 2013.

[21] 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.

[22] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic mapping studies in software engineering. In *12th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–10, 2008.

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

[24] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[25] Chanchal Kumar Roy and James R Cordy. A Survey on Software Clone Detection Research. Technical report, 2007.

[26] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. SourcererCC: Scaling code clone detection to big-code. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 1157—-1168, 2016.

[27] Yang Yuan and Yao Guo. Boreas: an accurate and scalable token-based approach to code clone detection. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, pages 286—-289, 2012.

Appendix A: Listing and Classification of All CDTs Reviewed by Bellon2007, Roy2007, and Rattan2013

Listing and Classification of All CDTs Reviewed by Bellon2007, Roy2007, and Rattan2013

CDT/Author	Bellon2007	Roy2007	Rattan2013	Classification in this report
duploc	[4] text	[74] text	[56] text	text
Johnson	[13] text	[118,120] text	[104] text	text

dup	[1] token	[18] text, token	[7] token	token
CCFinder	[9] token	[122] token	[113] token	token
Cordy	[16] token	[56] text	[41] text	text
Sim	[17] token	[90] token	[67] tree	token
Kontogiannis	[2] metrics	[146] metrics	[132,133] metrics	metrics
CLAN/Covet	[3] metrics		[144] metrics	metrics
CLAN	[12] metrics	[178] metrics	[170] metrics	metrics
DiLucca	[20]metrics	[67] metrics	[161] metrics	metrics
Lanubile	[21] metrics	[161] metrics	[146] metrics	metrics
CloneDR	[8] tree	[31] tree	[22] tree	tree
Yang	[22] tree	[222] tree	[231] tree	tree
Duplix	[10] pdg	[156] pdg	[138] pdg	pdg
PDG-DUP	[11] pdg	[141] pdg	[130] pdg	pdg
Marcus	[23] other	[177] text	[168] text	text
Leitao	[24] other	[164] hybrid		hybrid
CloneDetection	[25] other	[213] tree	[226] tree	tree
cp-miner	[26] other	[168,169] token	[152] token	token
RTF		[24] token	[16] token	token
ccdiml		[35] tree	[19] tree	tree
Asta		[77] tree	[58] tree	tree
gplag		[165] pdg		pdg
Patenaude		[190] metrics	[179] metrics	metrics
Dagenais		[58] metrics	[42] metrics	metrics
Buss		[45] metrics		metrics
Davey		[60] metrics		metrics
clones/cscope		[153] hybrid	[134] token	token
Tairas		[206] hybrid	[214] tree	tree
Greenan		[98] hybrid		hybrid
Deckard		[116] hybrid	[100] tree	tree
Similar Methods Classifier		[22] hybrid	[8] metrics	metrics
Simian			[207] text	text
DuDe			[228] text	text
SDD			[149] text	text
CSeR			[95] text	text
NICAD			[191] text	text
EqMiner			[102] text	text
Barbour			[13] text	text
D-CCFinder			[156,157] token	token
SHINOBI			[230] token	token
FCFinder			[199] token	token
Jian-lin			[103] token	token
Chilowicz			[35] token	token
SimScan			[208] tree	tree

Clone Digger			[31,40] tree	tree
ClemanX			[176] tree	tree
JCCD API			[24] tree	tree
cpdetector			[134] tree	tree
clast			[59] tree	tree
Chilowicz (tree-based)			[36] tree	tree
Saebjornsen			[196] tree	tree
Lee			[150] tree	tree
Brown			[27] tree	tree
Scorpio			[84] pdg	pdg
Choi			[37] pdg	pdg
Horwitz			[85] pdg	pdg
Li			[154] metrics	metrics
Antoniol			[3,4] metrics	metrics
Kodhai			[129] metrics	metrics
Perumal			[180] metrics	metrics
Lavoie			[147] metrics	metrics
CloneDetective/ConQAT			[105,106] model	model
ModelCD			[49,182] model	model
DuplicationDetector			[155] model	model
Mqlone			[209] model	model
Clone Detective			[47] model	model
Hummel			[90] model	model
Clone Miner			[17] hybrid	hybrid
MeCC			[125] hybrid	hybrid
Maeda			[166] hybrid	hybrid
Lucia			[164] hybrid	hybrid
Li (hybrid)			[153] hybrid	hybrid
Hummel (hybrid)			[89] hybrid	hybrid
Sutton			[210] hybrid	hybrid
Cordy (hybrid)			[74] hybrid	hybrid
DL_Clone			[202] hybrid	hybrid
Corazza			[40] hybrid	hybrid