Detecting Clones across Microsoft .NET Programming Languages

Farouq Al-omari[±], Iman Keivanloo[*], Chanchal K. Roy[±], Juergen Rilling[*]

[±]Department of Computer Science
University of Saskatchewan
Saskatoon, Canada
{faa634, ckr353}@mail.usask.ca

[*]Department of Computer Science
Concordia University,
Montreal, Canada
{i_keiv, rilling}@cse.concordia.ca

***Abstract*—The Microsoft .NET framework and its language family focus on multi-language development to support interoperability across several programming languages. The framework allows for the development of similar applications in different languages through the reuse of core libraries. As a result of such a multi-language development, the identification and traceability of similar code fragments (clones) becomes a key challenge. In this paper, we present a clone detection approach for the .NET language family. The approach is based on the Common Intermediate Language, which is generated by the .NET compiler for the different languages within the .NET framework. In order to achieve an acceptable recall while maintaining the precision of our detection approach, we define a set of filtering processes to reduce noise in the raw data. We show that these filters are essential for Intermediate Language-based clone detection, without significantly affecting the precision of the detection approach. Finally, we study the quantitative and qualitative performance aspects of our clone detection approach. We evaluate the number of reported candidate clone-pairs, as well as the precision and recall (using manual validation) for several open source cross-language systems, to show the effectiveness of our proposed approach.**

***Index Terms*—cross-language clone detection, intermediate language, binary, multi-language, similarity component.**

I. INTRODUCTION

Large software systems contain 10-15% of duplicated code which is also referred to as code clone [1]. Clones occur due to several reasons, such as software developers intentionally practice cloning to save time during software development, especially when reusing complete functional units. On the other hand, developers also unintentionally produce clones, by re-implementing similar functionality (code fragments) that already exists in the same system or another existing system.

In general, code clones are considered harmful and can reduce the quality of software [2, 3]. For example, when a modification is performed to a cloned fragment, all other instances of this fragment may require the same modification. This task duplication requires not only additional maintenance effort but also results often in additional cost for the software project. On the other hand, not all clones are considered as bad smells. Some cloning patterns can benefit development and maintenance [1] and are therefore sometimes even considered as essential or unavoidable [4].

As software systems become larger, more complex, and are being developed using more than one programming language, clone management becomes an essential part of the maintenance process. One approach to clone management is the use of clone detection tools, which discover the presence or absence of clones in a software system. While single language clone detection has been widely addressed and matured [5], only limited tool support exists for clone detection in multi-language/cross-language software development.

More recently there has been an ongoing trend towards multi-language software development, to take advantage of different programming languages [6] specifically in the .NET context. For multi-language development, two key usage scenarios can be distinguished: (1) combining different programming languages within a single, often large and complex system, and (2) use of several languages for re-implementation of a current system to support new client, application, or due to non-technical reasons (e.g. [7]). As a result, the ability to detect and manage similar code reuse patterns that might exist in these multiple languages systems becomes essential.

Over the last decade a variety of detection tools have been introduced [5, 8] which are typically based on parsing Abstract Syntax Trees (AST) or dependency graphs. These techniques use different matching approaches such as: string (token) similarity, vector similarity, sub-graph isomorphism or frequent set [8]. In addition, some clone detection tools do not rely on source code. They [9, 10, 11] instead use bytecode or intermediate representations as their input.

While many clone detection tools are capable of supporting different programming languages, they lack actual cross-language support during detection time. Consequently, these tools only detect clones in one program language at the time, and do not detect clones that span over multiple programming languages. Few studies have been conducted on multi-language clone detection. For example, in [12] a unified representation for the .NET is used to detect clones between VB .NET and C# at source code level.

Unlike early efforts, in this paper we focus on detecting clones across all Microsoft .NET programming languages (C#, C++ .NET, Visual Basic .NET, J# and F#) using Microsoft's Common Intermediate Language (CIL), as an intermediate representation of the disassembled binary .NET content.

In this research, we first highlight the importance of cross-language clone detection for Intermediate Languages. Second, since CIL is a low level human readable language, one has to deal with a larger amount of code which leads to additional textually variations and therefore new detection challenges. We established several preprocessing steps for the CIL code to optimize it for clone detection and to reduce its content dissimilarity. Applying these different filtering and optimizations techniques allows us to improve the overall accuracy and performance of the clone detection. As a result our research addresses several fundamental research questions which are listed below.

RQ1: *Are "the selected filters" useful?* In this experiment we observe how much it is likely that the filtering approach contributes to the true positive ratio.

RQ2: *Are "the filters" a major threat to the precision by making non-cloned fragments similar to the actual clone pairs (because of filtering some critical data)?*

RQ3: *Is our "clone detection approach" able to detect cross-language clones on .NET using IL?*

RQ4: *How successful is the "clone detection approach" in terms of precision and recall?*

We conducted several case studies on four datasets created from open source software systems written in C#, J#, and VB.NET. We used one of the datasets as our oracle for objective recall measurement. We applied three clone detection algorithms to avoid algorithm-dependent observation as much as possible. We manually investigated ~2K clone-pairs (randomly selected) to measure our approach's precision and recall. Finally, we observed that our approach is able to detect high quality cross-language clones successfully at method level granularity using Intermediate Language.

The remainder of the paper is organized as follows. In section 2 and 3 we provide the motivation and background about CIL and adopted clone detection tools and algorithm. Section 4 describes our proposed process and the filter set details. Section 5 studies the necessity of using filters (RQ1). We investigate the potential filters' negative effect on precision (RQ2) in Section 6. Finally, in Sections 7, 8 and 9 performance evaluation (RQ3 and 4), related work, and paper conclusion are presented.

II. Motivation - The Necessity for Unified Representation

In order to evaluate the necessity of using a unified representation (e.g., CIL or Kraft et al. [12] approach) as an intermediate representation for cross-language clone detection, we conducted some case studies. The objective of these studies is to establish a comparison between selected unified source code representation (CIL in the context of our research) and source code-based clone detection for cross-language clone detection.

For the case study, we adopt a language-independent comparison engine (the clone detection tool). Second, we feed source code from different languages to the function. Finally, we repeated the clone detection process on the corresponding CIL content. We analyze the effect of the cross-language clone detection, by replacing CIL with the actual source code written in different programming languages. We conducted a quantitative study to evaluate and compare the detection results. For the study we detect cross-language clones in the different Mono compiler [13] versions (implemented either in C# or VB.NET), as well as the ASXGUI [14] C# and VB.NET versions. We then analyzed the reported clone clusters for both CIL and source code (Table 1).

TABLE I. Comparison between clone detected using CIL or source code as input

Dataset	Input Data Type			
	CIL		*Source Code*	
	# Clone Class	*# Clone Fragment*	*# Clone Class*	*# Clone Fragment*
ASXGUI	9	393	69	261
Mono	37	4373	369	1523

Observations. Compared to source code, using CIL more cloned code *fragments* can be detected. Second, our manual validation of the reported clones showed that many of the missed clones at the source code level were actually *near-miss* clones, with these clones containing more than one line differences. In conclusion, CIL based clone clusters always contained a larger number of code fragments with low cohesion. However, the overall recall and precision at the clone-pair level (not clone class) is higher compared to a source code approach and therefore shows that an intermediate language improves clone detection in a cross-language setting.

III. Background

In this section, first, we provide an overview of Microsoft's .NET Common Intermediate Language (CIL). Second, we review the code clone concept and the comparison algorithms used for this study.

A. Common Intermediate Language (CIL)

The .NET Framework[1] is a software development platform developed by Microsoft that runs primarily on Windows. It consists of several components such as, (1) a comprehensive library of commonly used functionalities, (2) a runt-time management environment (language independent), and (3) a set of programming languages. Contrary to Java, which targets application development using one language on several platforms, .NET aims for multi-language development on a single platform. It provides language interoperability, with each program module being able to use code written in the other languages.

The source code of various .NET programming languages (C#, Visual Basic .NET, Visual C++ .NET, J#, F# etc.) is compiled into the CIL. When the .NET managed code is compiled, the compiler first converts it into Common Intermediate Language (CIL), a machine independent

[1] http://www.microsoft.com/net

intermediate language, before compiling it into .NET portable executable (PE). Visual Studio SDK includes a disassembler "ildasm.exe" that takes the Portable Executable (PE) file(s) and generates the CIL in human readable formats such as plain text (e.g. Fig.1–Column #2).

CIL is an object-oriented, stack-based like assembly language. It is also referred to as Microsoft Intermediate Language (MSIL) or Intermediate Language (IL). CIL is a platform independent language that can be executed in any environment that supports the Common Language Infrastructure. CIL itself is also a .NET programming language, which can be in combination with the CIL .Net compiler "ilasm.exe" also be used directly to develop applications in CIL.

Examples and Challenges. Being a lower level representation, CIL code size tends to be much larger than traditional high-level source code. Figure 1 (the first two columns) shows a comparison between a VB code fragment (a small VB method), and its corresponding CIL representation. In this example the method body with five lines of code has been transformed to more than twenty lines of code in CIL. This creates an additional challenge, making clone detection on binary rather different from source code.

Nevertheless, given this common representation of code fragments written in different programming languages provides the ability to use CIL for clone detection across .NET languages. However, a key challenge is the fact that it is possible to have some dissimilarity at CIL level, even in cases of semantically identical source code fragments (written in different .Net languages). The first four columns of Fig. 1 (the Raw Data section) provide an example for such dissimilarities. Both the VB and C# methods implement the same program following similar coding pattern and structure as much as possible. However, when we compare the CIL pairs, there are three key sections clearly distinguishable: (1) identical CIL content which is marked by the first dashed area, (2) the first point of dissimilarity which is flagged by the italic font style, and (3) the rest of the content marked by the second dashed box that covers CIL content with considerable dissimilarity. In general, this example highlights the *key* challenge in binary clone detection, the possibility of facing dissimilarity by exploiting .NET Intermediate Language even for semantically (and almost syntactically) identical fragments in cross-language context.

B. Source Code Clones

Two code fragments that share some degree of similarity are typically considered a clone pair. Based on their actual similarity, clone pairs can be categorized [5, 8] as *Type-1*, *Type-2*, *Type-3*, and *Type-4* clones. Type-1 clones are exact copies of each other, except for possible differences in whitespaces, layouts and comments. Type-2 clones are syntactically identical fragments except for variations in identifiers, literals, data types, whitespace, layouts and comments. Copied fragments (e.g., Type-1 and Type-2 clones) with further modifications such as additions, deletions and changes of statements are called Type-3 clones. Type-2 and Type-3 clones are also known as near-miss clones. Code fragments that perform the same computation (e.g., semantically similar) but implemented through different syntactic variations are called Type-4 clones. Note that all of these definitions were originally introduced for clone-pairs implemented in the same programming language. In our cross-language clone research these definitions are no longer applicable *as-is*, and have to be refined to meet our research context. For example, the VB and C# fragments in Fig. 1 would be considered Type-1 clones in the cross-language clone detection since they are essentially performing the same task implemented in different programming languages.

C. Clone Detection Algorithms

In our research we use SimHash [15], Longest Common Subsequence (LCS) [16], and Levenshtein Distance [17] algorithms to detect clone-pairs. Note that the first two techniques are adopted from SimCad [18] and NiCad [19] respectively. Our primary research goal is to address clone detection challenges in the .NET Intermediate Language by providing solutions which are general applicable and independent of a specific clone detection algorithm. We therefore selected three dominant algorithms for modeling *edit distance* in this paper to avoid an algorithm specific solution.

1) LCS-Based Clone Detection

The Longest Common Subsequence (LCS) [16] algorithm detects the longest common subsequence between two strings. For example, consider the following two sequences of characters.

S1 = AABBBCDABCDDAABD
S2 = DDABCCCDAABBBDAC

For the above example, the LCS among the S1 and S2 sequence is ABCDABDA. For our cross-language clone detection we applied LCS to the CIL instruction sequences in order to determine the LCS similarity of CIL code fragments (e.g. highlighted part in Fig. 1). The LCS size is important since it is the similarity measure we use to decide if two fragments form a candidate clone-pair in our research.

2) Levenshtein Distance-based Clone Detection

Levenshtein Distance (LEV) [17] is one of most widely used algorithms to calculate the *edit distance*. Contrary to LCS (where the actual output is a string), LEV provides as output the dissimilarity between two string sequences as a single number value. We use *LevSim* (Eq. 1) and its output is compared against a constant threshold value, to decide whether two fragments (i.e. S_1 and S_2) will be reported as candidate clone-pairs.

$$LevSim(S_1, S_2) = 1 - \frac{\text{LEV}}{\max(size(S_1), size(S_1))} \qquad (1)$$

3) SimHash-based Clone Detection

SimHash algorithm constitutes the core of SimCad. It generates a 64-bit fingerprint, which we use to detect clones based on their fingerprint similarities. The algorithm uses Charikar's [20] hash function where the Hamming Distance is used as the crucial configuration parameter.

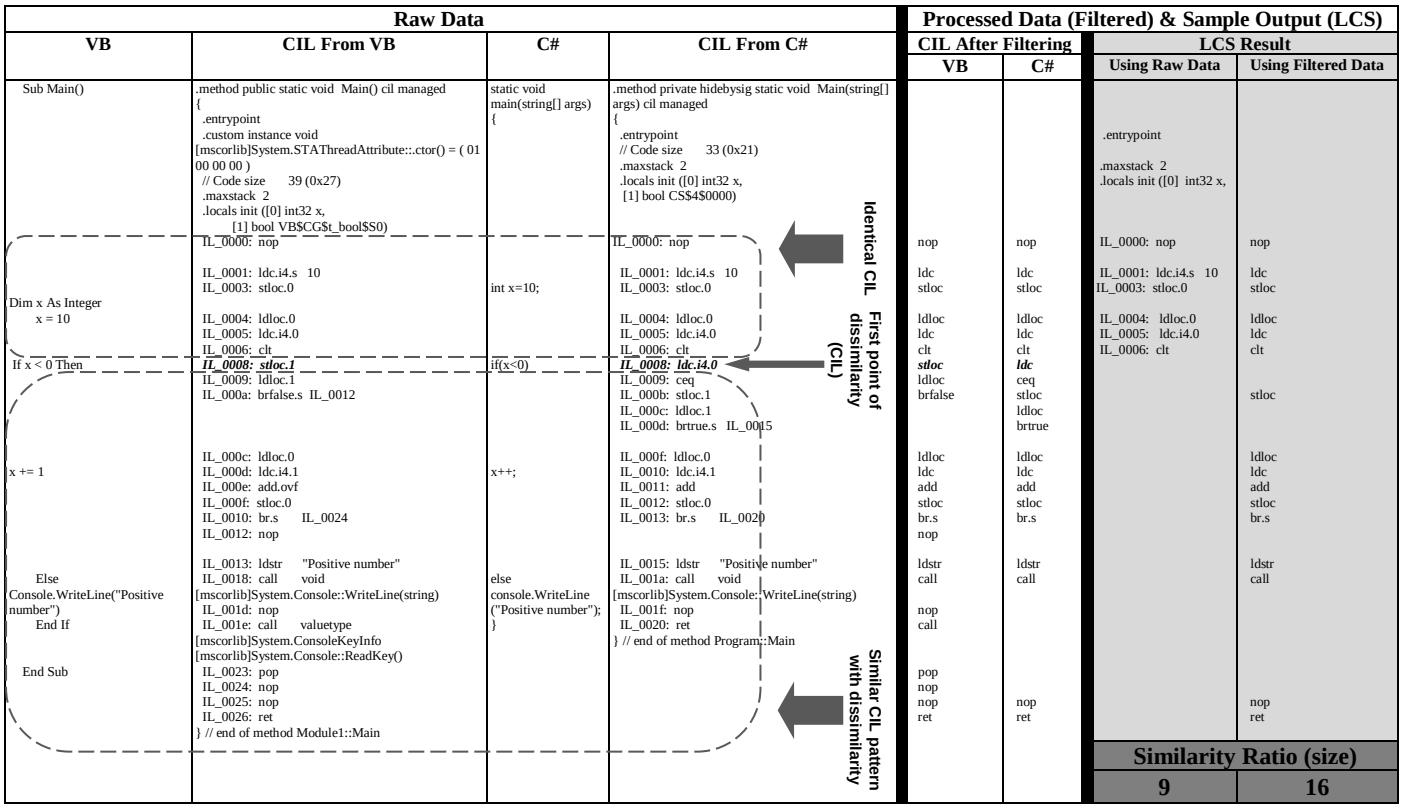

Fig. 1. First Part (Left Section): A C#/VB methods including their corresponding CIL – The example shows the challenges in clone detection using Intermediate Language (e.g., larger size, unexpected dissimilarities in CIL). Second Part (Right Section): An example to show how our filtering proposal contributes to the clone detection by improving the similarity (e.g., LCS) between fragments.

The Hamming Distance represents the number of positions at which the corresponding bits are different between two fingerprints. A Hamming distance of zero corresponds to identical fingerprints and therefore also to Type-1 clones, while a Hamming distance larger than zero reflects near-miss clones.

IV. Clone Detection Process (across .NET Languages)

Figure 2 shows the overall processing steps for detecting clones across Visual Studio .NET programming languages. First, the .NET language source code is collected and compiled, to obtain the .NET portable executable files. During the second processing step, the disassembling of corresponding executable files takes place through the Microsoft's Intermediate Language Disassembler (ildasm.exe). The disassembler generates the CIL code as plain text files. The CIL files are then parsed to extract all method/function bodies. In the next step, we apply our filters on the CIL code. They play a key role in our approach since they eliminate undesirable noise and improve the overall quality of the CIL files. After applying the filters on the CIL, we run the selected comparison algorithms to detect clone-pair candidates. As part of this clone detection phase, we use, (1) LCS, (2) Levenshtein Distance and (3) SimHash-based algorithms. Finally, a clone-pair report will be generated for both CIL and source code (by mapping the CIL clones to their corresponding source code).

A. The Motivation for the Filter Set

Figure 1 shows an example of a program snippet written in C# and VB including their corresponding CIL representations. The key challenge (as discussed earlier): although both methods implement the same program and are compiled into the same Intermediate Language, still there is some dissimilarity in their intermediate representations which can affect the clone detection process. We address this challenge by creating a set of cleaning and filtering steps for CIL to improve the performance of Type-1, Type-2, Type-3 and Type-4 clone detection in the CIL code. The filters are designed to improve the detection rate (i.e., recall) since the CIL data contains a significant amount of noise (e.g., reference numbers to string tables, which are compilation context dependent). Due to such noise in the CIL files, two semantically identical source code fragments might no longer be considered as highly similar at the CIL level (e.g., content similar VB and C# methods might have less than 50% similarity at the CIL level, see Fig. 1).

B. Overview of the Proposed Filters

Prior to the actual filtering process a pre-processing step takes place that removes all directives (e.g., `.method`, `.entrypoint`, and `.maxstack`) that appear at the beginning of the CIL representation of a particular method. After this preprocessing step, the actual CIL code sequences are analyzed and filtered accordingly. In what follows we provide a summary of the eight filters we create (Table 2) and provide an example for each filter.

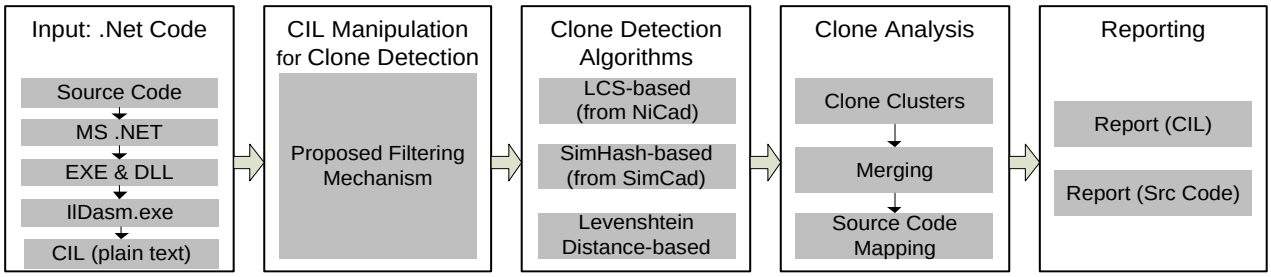

Fig. 2. Schematic diagram for the proposed cross-language clone detection and result evaluation

TABLE II. EXAMPLES OF CIL FILTERS

	Before Filtering	After Filtering	Example Description
Filter 1	**IL_0003:** stloc.0	stloc.0	Where IL_0003 is the instruction address
Filter 2	brtrue.s **IL_0015**	brtrue.s	The IL_0015 address of the branch destination
Filter 3	ldarg **3** starg **1**	ldarg starg	The value 3&1 represent argument number
Filter 4	ldc.i4.s **10**	ldc.i4.s	10 is the number (pushed to the stack)
Filter 5	ldstr **"Positive number"**	ldstr	"positive number" is the printed string constant
Filter 6	stloc **7**	stloc	7 represents variable index
Filter 7	ldc**.i4.s 10**	ldc	i4 represent the int32 data type in CIL and s for Short
Filter 8	**IL_0011:** add **IL_0012:** stloc**.0** **IL_0013:** br.s **IL_0020** **IL_001a:** call **void [mscorlib]System.Console: :WriteLine (string)**	add stloc br call	Note that Filter 8 is just a nick name. Refer to the Filter 8 description section for more details.

Filter 1: Removal of the instruction address (IL_xxxx:) at the *begin* of each CIL instruction, eliminating dissimilarities due to application/environment specific variations.

Filter 2: Removal of instruction address (IL_xxxx:) for *branching statement*. As part of this filtering step we cover all 33 branching statements (e.g. `beq`, `beq.s`, `bge`).

Filter 3: Removal of integer values that represent argument number in CIL. e.g. ldarg 3 is interpreted in CIL as load the argument number 3 onto the stack. Instructions included in this filter are: `starg`, `starg.s`, `ldrag`, `ldrag.s`, `ldrags`, and `ldraga.s`.

Filter 4: This filter eliminates *constants* in the CIL code, e.g. "ldc.i4 num" which corresponds to a *Push num* of type *int32* onto the stack as int32. Instructions covered by this filter are `ldc.i4`, `ldc.i8`, `ldc.r4`, `ldc.r8`, and `ldc.i4.s`.

Filter 5: This filter removes all print literals in the CIL code, which are identified through `ldstr` statements.

Filter 6: This filter removes all variable indexes like *stloc index*, which correspond to popping a value from stack into a local variable. Among the instructions removed by this filter are: `ldloc`, `ldloc.s`, `ldloca.s`, `stloc` and `stloc.s`.

Filter 7: This filter removes some additional data types and constant integers such as *i4* from "ldc.*i4. 1*". The complete command pushes 1 as an int32 onto the stack.

Filter 8: Is not actually a new filter, it combines all seven filtering techniques mentioned above, including the preprocessing tasks in one single filter.

V. FILTERS' CONTRIBUTION EVALUATION

As discussed earlier, we propose this *filter set* to increase the recall (by reducing noise and dissimilarity in the CIL code) and to be able to detect other valuable clone types such as type-3 clones. For example in Fig. 1, we were able to successfully increase the similarity ratio of the clone-pair from 9 (before filtering) to 16 (after applying filtering). In order to answer in more detail our first research question (RQ1), we conduct an experimental evaluation to determine how much our filtering approach actually contributes to the true positive ratio and its benefits to the overall detection process.

To answer this question, we defined a metric called *Filter Contribution* that measures the effectiveness of each filter. The underlying idea is to measure the similarity degree of candidate clone-pairs before and after applying different filters. The measure will indicate how much a particular filter increases the similarity value between two fragments. Note that in the ideal case, we expect that a filter would increase the similarity values of true positive cases significantly more than the ones for false positive cases. Otherwise, a particular filter would not be useful to discriminate (with high confidence) against false positives. The Filter Contribution (*FltrCntrb*) function is defined in Eq. (2), which is based on LCS-based similarity. S_i denotes the participant fragments in the clone-pair under investigation and F_x presents the filter function with *x* being the filter number.

$$Sim(S_1, S_2) = \frac{size(LCS)}{[size(S_1) + size(S_1)]/2} \qquad (2)$$

$$FltrCntrb(F_x, S_1, S_2) = Sim(F_x(S_1), F_x(S_2)) - Sim(S_1, S_2)$$

The challenging part of this experiment was to identify proper input data, due to external constraints caused by the availability of .NET source code on the Internet. More specifically, the challenge was that we had to identify similar systems written in more than one .NET language. Fortunately, the iText.NET package [21] (note that it is different from iText project [7]), met our input constraint. Although iText.NET is originally written in J#, it includes C#, VB, and J# methods for its 25 major use cases. Therefore, our first dataset (a.k.a. *Cloned Fragments Dataset*) contains 25 clone classes, with each clone class having three clone fragments, with a code fragments being implemented using in three different .NET programming languages (C#, VB and J#). Note that we mutually created three true positive clone pairs by following the VB-C#, VB-J#, and C#-J# patterns for each use case using the iText.NET API usage code [21] (Example Code Section).

This approach resulted in 75 distinct clone pairs (i.e., actual true positive clone-pairs). The second dataset (a.k.a., *Non-cloned Fragments Dataset*) contains 25 non-clone classes and 75 false positive clone-pair candidates created in the same manner as clone classes. We want both tagged datasets to be able to answer RQ1 (i.e. whether filtering has any positive/negative effects on cross-language clone detection).

We then measured the *Filter Contribution* value for each filter when applied on both datasets. The result for each data set is shown in Fig. 4 and Fig. 5. We excluding Filter 8, since this filter does not introduce a new threat (i.e., no negative effect) to our clone detection approach, since this filter only engages all other filters. In most cases the filters increased the similarity up to ~0.2 (max) for non-cloned pairs while improving the similarity of cloned pairs by at least ~0.3. This result supports our research hypothesis that filtering increases the similarity values for true positive cases (the cloned dataset) with a higher ratio than the false positive cases (the non-cloned dataset). Comparing the results of Filter 8 between Fig. 4 and 5, it is observable that the answer to RQ1 is positive since the overall contribution of the filters improves the similarity degree on actual cloned pairs much more than for non-cloned fragments (non-cloned pairs less than 0.5, while for the majority of cloned pairs the similarity increases between 0.5 and 0.8.

To support our claim, we conducted another case study on the same dataset to determine if our filters can be used to identify an appropriate similarity threshold. Figure 3 summarizes the findings, showing that before applying our filters, there was no clear distinction between similarity values of actual clone-pairs (true positives) and false positives. Therefore it is impossible to determine an adequate threshold that allows separating actual clones from false positives. In contrast, Fig. 3 shows that filters address this problem by increasing the distance between the two groups (tagged on the right side of Fig. 3). For example, using our filters, a threshold from 0.4 to 0.55 can separate true positives from false positives with high confidence. Our analysis therefore supports the usefulness and necessity of the proposed filter set (RQ1) for cross-language clone detection on CIL.

VI. Does Filtering Make Actual Clone-pairs and Non-cloned pairs Similar?

A major threat to any filter-based approach is the loss of precision by filtering out essential data. As a result, excessive or improper loss of data (due to filtering) can lead to situation where *non-answers* and *actual answers* become similar to the decision making algorithm, which eventually leads to an increase in the false positive ratio.

In this research, we are interested to observe the similarity between actual clones and non-cloned fragments after filtering (to answer RQ2). Apparently, we prefer to have fewer similar entities when we compare members of actual clone-pairs with other paired fragments (i.e. non-cloned fragments) to support the applicability of our filtering approach.

In order to observe the similarity/dissimilarity of code fragments we adopt the Chernoff face [22] visualization approach. The Chenoff face visualization represents data as glyphs similar to human faces while each dimension is being mapped to a specific feature of the face (e.g., the Filter #5 similarity contribution value determines the distance between eyes for each clone-pair). We application of glyphs in software visualization is a well-established research area (e.g. [23]).

For our controlled experiment, we produced seven face features for each pair by calculating *Filter Contribution* on all seven filters separately. That is, each pair can be modeled using a vector in a multi-dimensional space (in our case, seven dimensions). Based on this assumption, two sub research questions arose (RQ2 breakdown): (RQ2-a) "do filters make actual clone-pairs similar to non-cloned pairs?". In other words, "is the filtering approach misleads the clone detection approach to report a false positive as clone-pair?" or "is filtering a major threat to false positive ratio of cross-language clone detection using CIL?", and (RQ-b) "is filtering neutral to the participating programming languages of clone-pairs (in cross-language clone detection context)?".

Figures 6(a) and 6(b) show the result for two datasets (similar data as in Section IV). By comparing the faces between Figs. 6(a) and 6(b), it is possible to answer the RQ2-a: filtering does not make non-cloned pairs similar to actual clones. Therefore filtering becomes not a major threat for the precision in our research. For example, we can observe that there are more distorted and super tiny faces (i.e., pairs - e.g. the second face of Fig. 6(*a*)) available in Fig. 6(a) which contains non-cloned pairs than Fig. 6(b). The issue can be attributed to Filter 1, 2, and 5 since they are mapped to: (1) the face size, (2) distance between forehead and jaw, and (3) distance between eyes respectively. Therefore, it is also possible to intuitively observe that Filter 1, 2, and 5 (including Filter 7 observed in Fig. 5) play the major role in characterization of true positives.

To answer RQ2-b, we categorized the clone pairs based on the programming language. Figs. 6(c) and 6(d) illustrate the result. For example, the top category in Fig. 6(c) contains all pairs where the first fragment is written in VB and the other fragment is in C#. As it is obvious C#-J# pairs in Fig. 6(d) (cloned pairs) are different. That is most of the faces are not *round shaped* comparing to the two other groups in Fig. 6(d). The same pattern is observable in Fig. 6(c) C#-J# category which contains mostly non-rounded shaped faces. Therefore, we can confirm RQ2-b, filters are independent of the programing language. This observation can be attributed to the resemblance between C# and J# languages and their history where the IL content becomes highly similar where we filter out the line numbers (i.e., Filter 1 and 2).

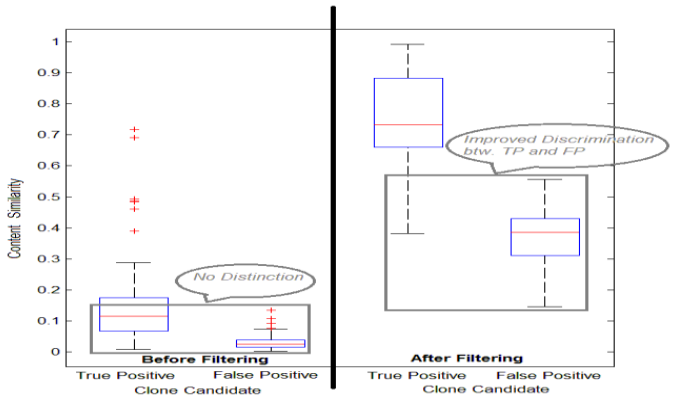

Fig. 3. Filters' contribution to discriminate between actual clone pairs and non-cloned pairs

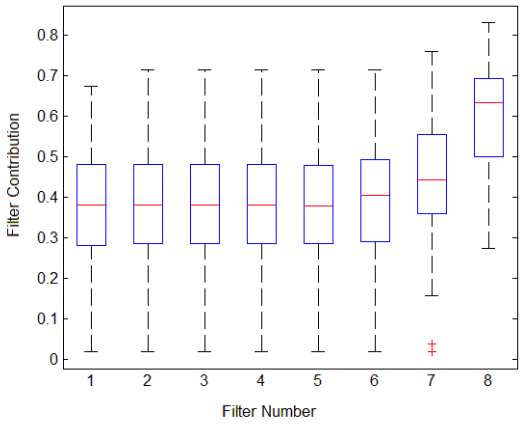

Fig. 4. Filtering Effects on the *Cloned* Dataset

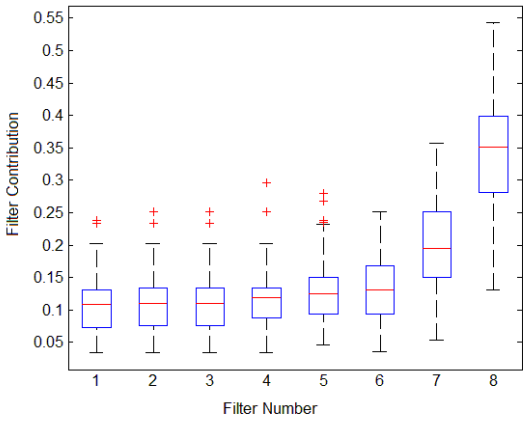

Fig. 5. Filtering Effects on the *Non-cloned* Dataset

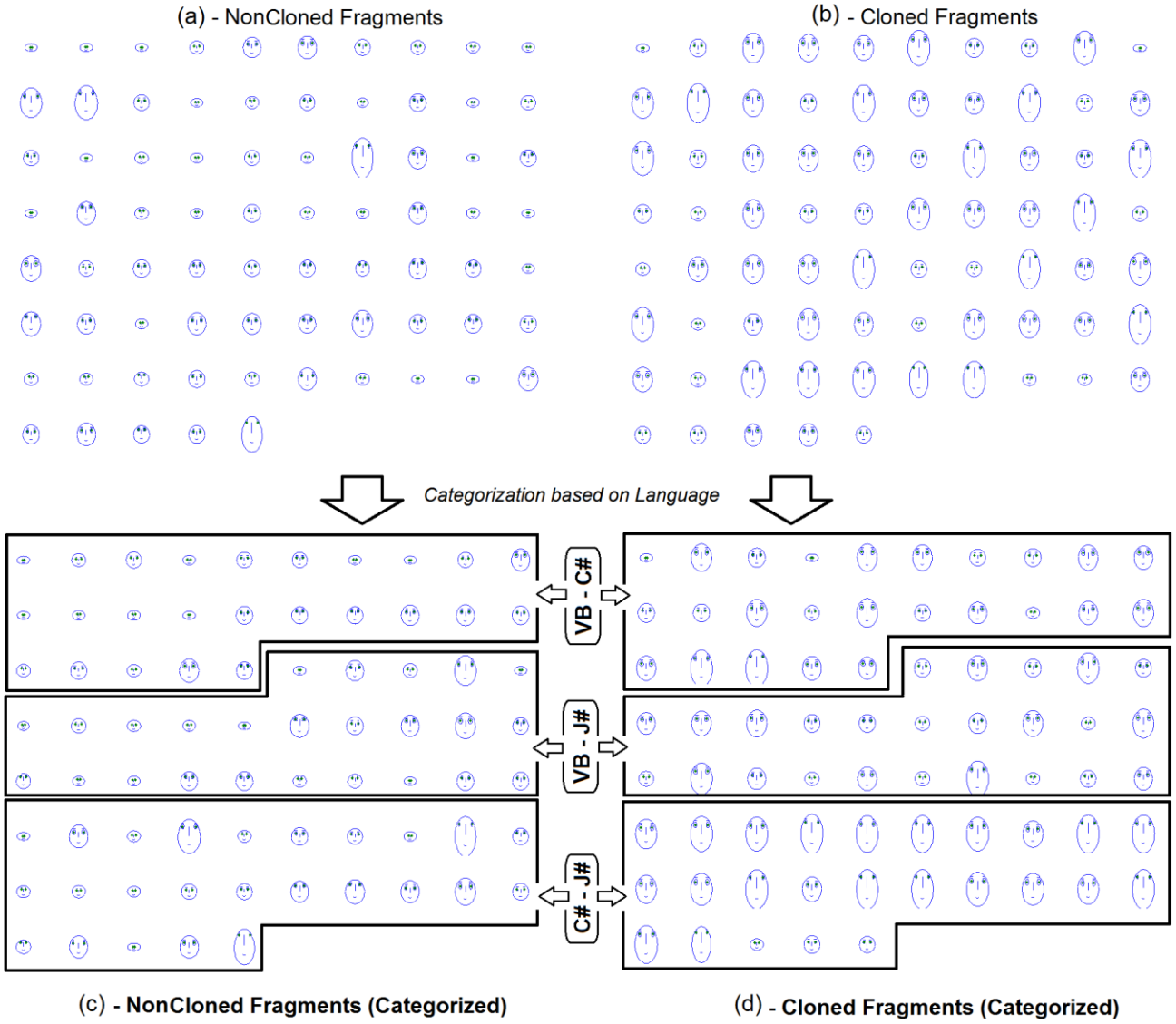

Fig. 6. *Filter Contribution* data are mapped to multi-dimensional space to investigate the importance of filters and language dependency using glyph visualization. Note that numbers are used for referencing purposes in this figure (i.e. there is no ordering relation between faces etc. specifically between faces in the left and right hand sides – in short each face represents just one candidate clone-pair)

VII. EVALUATION

In this section, we present the evaluation results from our cross-language clone detection using Intermediate Language on four .NET systems. We analyze the clone detection results from a qualitative and quantitative perspective, using three edit distance methods (LCS, LEV, SimHash) to answer RQ3 and RQ4. Note, all datasets used for the evaluation include at least two .NET languages.

The first dataset contains two versions of the ASXGUI [14] open source GUI encoder. The dataset contains Version 3.0 and 2.5 since its current version (Ver. 3.0) is developed in C#, while the earlier implementation was based on VB.NET. ASXGUI Ver. 2.5 consists of 47 VB.NET files with a total of 32594 lines of source code and 303 functions. ASXGUI v 3.0 on the other hand, consists of 19 C# files with 2088 lines of source code, and 78 functions. The combined number of files being analyzed is 66 files with a total of 34682 lines of source code. The noticeable difference in project metrics (e.g., LOC) can be attributed to the (1) dissimilarities in the programming languages, and (2) re-engineering and refactoring tasks.

The second dataset is based on the C# and VB.NET compilers from Mono [13], version 2.10. The C# compiler consists of 57 C# files and the VB.NET compiler of 375 files, with a combined number of 432 files and 4998 functions.

The other two systems used in iyr study are two PDF libraries called iText and iText.NET. While their project names are similar, both projects are completely independent from each other. We created our third dataset from the iText (C# branch) and iText.NET (J#) source code. The dataset contains more than 600K LOC and 2.5K files.

We used part of iText.NET library to create our last dataset. This dataset contains source code related iText.NET API usage written in three languages (C#, J#, and VB.NET). This feature makes the dataset an important resource for our study since it allowed us to create a small (75 clone pairs) but controlled dataset (i.e., all actual clones are aligned, tagged and known in the cross-language), creating a unique oracle for further analysis. We use this oracle to obtain precise recall and precision measures, since the number of actual clones is known. This is contrast to the other datasets, where recall and precision measure cannot be computed as precisely, since the actual number of clone-pairs is unknown.

A. Quantitative Evaluation

Figure 7 shows the total number of detected candidate clone-pairs from the filter datasets using the three selected distance measure algorithms. The results from this experiment can be summarized as follows: (1) it is possible to detect numerous candidate clone-pairs even for cross-language case regardless of the underlying algorithm, (2) no candidate clone-pair is detected for cross-language using 1.0 as the Similarity Factor (i.e., the decision making threshold), which would only report clone-pairs with complete identical content. Therefore, even using *filtering* on highly similar cross-language clone-pairs (e.g., Fig. 1), some dissimilarities will have to be handled by by the clone detection approach. However, this is not the case for single language clone detection (shown in Fig. 7), (3) for all dataset, we can observe a major decrease in the number of candidates when the threshold value is set to a range between 0.6 and 0.8 (marked by ovals). Therefore, we can conclude that the detected range can be recommended to the end-user to provide an acceptable recall and precision (the supporting argument is given in the qualitative evaluation section). The only exception is SimHash, which reports for the same thresholds a lower number of candidate clone-pairs. This is due to the fact that SimHas uses a different threshold schema compared to the other distance measure algorithms.

B. Qualitative Evaluation

Since a noticeable number of candidate clone-pairs have been detected (Fig. 7) using the *filtering* approach, it is necessary to evaluate the quality of our approach, by manually validating the results (RQ4). We manually examine candidate clone-pairs to determine whether they are true or false positives. We investigated three distinct thresholds from the selected range discussed in the previous section, which we refer to as Extreme (threshold = 0.6), High (threshold 0.7), and Normal (threshold – 0.8) configurations. The objective is to observe the best and worst achievable true positive ratio using the Normal, High and Extreme configurations. We also applied random sampling since the total number of candidates even for the chosen thresholds was considerably large (e.g., 10K clone-pair). Finally, we evaluated ~2K candidate clone-pairs which were selected randomly.

1) Challenges in Quality Assessment for Cross Language Clone Detection.

Quality evaluation is inherently challenging in our research since there is no clear agreement on what constitutes true positives (TP) and the various clone types definitions. Therefore, we applied in our qualitative evaluation the following approach: (1) since it is possible to easily locate with confidence false positives among candidate clone-pairs, we first tag all false positives; (2) we assume the rest as true positive. However, in order to provide a more in depth quality assessment, we also analyze the quality of the reported true positives. One of the interesting example which we identified in the ASXGUI dataset is the true positive shown in Fig. 8. For this example it easy to select an appropriate corresponding clone type based on the existing defacto clone definitions (e.g. [8, 5]). Regardless of dissimilarities introduced by different languages (e.g., VB.NET vs. C#), it is obvious that: (1) both methods in this example are implementing the same functionality, and (2) most importantly they are following the same algorithm. Therefore, although we cannot select the clone type, we consider such clone-pair candidates as *strong true positive* in our *qualitative evaluation*. Therefore, we consider all clone pairs similar to this example as *strong* TPs and label them with *E* , and the remaining TPs are being labeled as *S*.

2) Quality Evaluation Result

Figure 9 reviews the findings of our quality evaluation from manually assessing ~2K candidate clone-pairs (answering RQ4). In general, using the *Normal* threshold all candidate clone-pairs that were reported are true positive (100% TP). The quality decreases with less restrictive thresholds. For example using SimHash and the Extreme threshold, the reported TP reduces to ~40%. The optimum, considering the trade-off between precision and recall, was achieved using Levenshtein Distance-based comparison with the High threshold (80% TP). Nevertheless, this result is not 100% precise (threats to validity) due to the sampling process and data dependency.

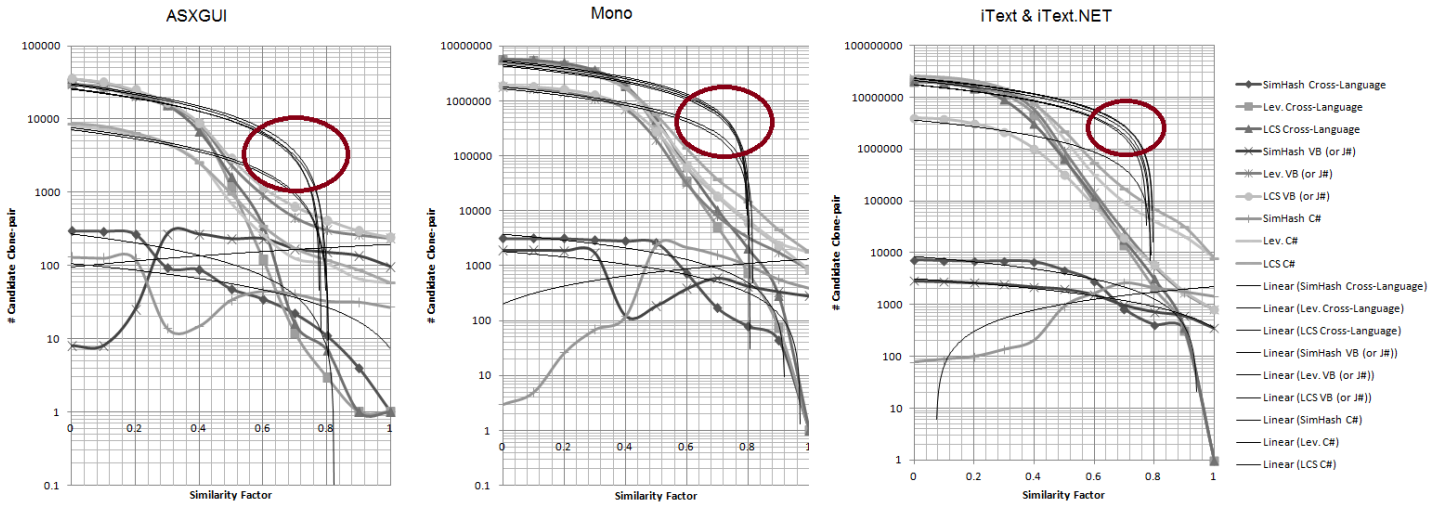

Fig. 7. Number of clone-pair candidates per (1) dataset, and (2) clone detection algorithm. Note that the Similarity Factor varies between 0 and 1. For example 1 is the strictest threshold which leads to detection of only exact content. The thin black lines show the linear trend for the corresponding case study. However they appear as curved lines since the horizontal axis is logarithmic on purpose. Following this approach, it is possible to observe the major drop area (threshold) in number of detected candidate clone-pair which is between 0.6 and 0.8 for almost all datasets and algorithms (with few exceptions).

*J# language only appears in iText & iText.NET dataset. +Note that the fluctuation in SimHash-based result is due to the internal SimCad logic

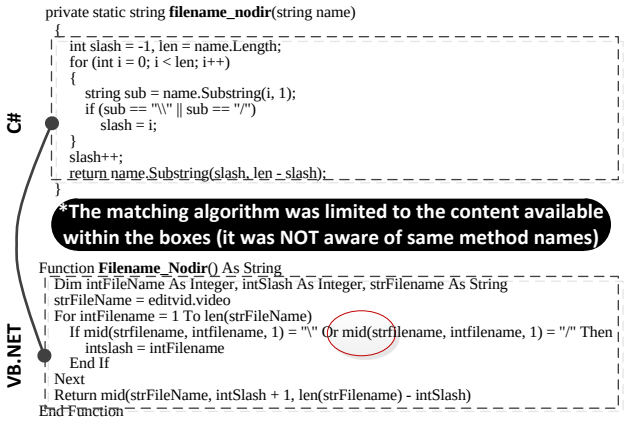

Fig. 8. An example of two strongly similar clone-pair detected by our approach from ASXGUI. Even in case of major dissimilarity such as occurrence of *mid* method in the VB section for several times, still our approach successfully detected the clone-pair.

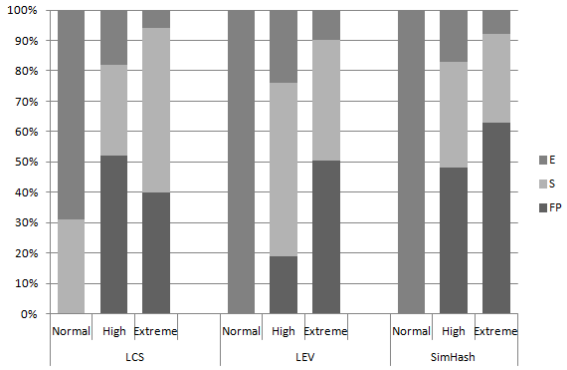

Fig. 9. Result of true/false positive (within candidate clone-pairs) evaluation using manual analysis. Normal, High, and Extreme stand for corresponding threshold from the selected range in the Quantitative Evaluation Section.

The other major aspect of our quality evaluation is the recall measurement (RQ4 recall section), which we calculated on our only available oracle (iText.NET API). In our evaluation, we observed a recall of 76% using High threshold between three

languages (C#, J#, and VB.NET). Note, we did not compute the recall for the other datasets, due to the lack of an objective assessment of what constitutes an actual TP for these dataset and consequently, would make any recall computation for these datasets prone to subjectivity.

VIII. Related Work

While there are numerous well-established clone detection tools available to support different programming languages [5, 8], there exist only very limited research related to (1) *cross language* or (2) *binary-level* clone detection. In our review of previous work we focus on these two research domains, since they are the closest related to our research. To the best of our knowledge, C2D2 [12] is the only tool capable of detecting cross-language clones. It uses NRefactory Library to generate the Unified CodeDOM graph for both C# and VB.NET. A string is generated by traversing this graph and targeted to string matching algorithm.

There are a few but diverse approaches on Intermediate Language-based clone detection (focusing on single language clone detection, mostly Java). One of the first studies on Intermediate Language clone detection is by Baker [10]. After some preprocessing (e.g., remapping offsets), she uses three comparison techniques (e.g. Diff [24]) to find similar fragments. Davis et al. [9] use the disassembler for both Java and C/C++ to detect clones in single language. They provide a public framework [25] for pretty-printing of disassembled code which constitutes the baseline of the clone detection phase. The most interesting aspect of their research is the proposed search algorithm for content matching which has two greedy and hill-climbing analysis steps. In [27] Selim et al introduce "Jimple" [26] to detect clones using an intermediate representation. The motivation is to exploit Jimple characteristics (comparing to stack-based Java native IL). Recently, Juricic [28] uses Intermediate Language code to detect plagiarism and similarities. The approach is based on Levenshtein Distance as the similarity measure to compare disassembled C# binary, and applies some primitive preprocessing techniques which are comparable to two of our filters. There are also some formal approaches, such as by Santone [29] that transform Java bytecode to mathematical models for clone detection.

To the best of our knowledge, our study presents the first comprehensive research focusing on, (1) .NET clone detection, (2) across programming languages, and (3) using Intermediate Language. Moreover, we not only proposed the approach, but also evaluated its major potential threats using diverse statistical and intuitive analyses. Finally, we evaluated its performance using manual validation to measure precision and recall. We observed a promising result in terms of both quantity and quality using three clone detection techniques for C#, J#, and VB.NET cross language clone-pairs.

IX. Conclusions and Future Work

With the globalization of the software industry and introduction of new programming languages, there has been an ongoing trend towards combining or re-implementing systems using different programming languages. This poses a new challenge for software comprehension, maintenance, clone management, and refactoring.

In this paper, we study a novel approach to detect cross-language clones in the Microsoft.NET Environment. In our research, we exploit CIL an intermediate representation generated by .NET compiler from all .NET programming languages. We established a *filter set* (containing 7 filters) which are applied to CIL prior to the actual clone detection process to remove noise in the dataset and establish threshold values for the detection algorithms. Using face glyphs, we showed that the filters do not remove crucial data from CIL and therefore have no negative effect on the clone detection. Finally, we performed a qualitative and quantitative study of a clone detection approach on four datasets. We used three widely used edit-distance functions to allow for a generalization of our observations and study results. In our evaluation, we observed that it is possible to detect a reasonable number (quantity) of clone-pairs with acceptable precision (based on the configuration) and recall (quality).

As future work, we will further expand our study to include additional platform. We also plan to investigate the use of other intermediate representations found in other frameworks, such as the unified bytecode generated by LLVM [30] compiler to detect clones across LLVM programming languages. Moreover, we are going to start a comprehensive usability study using our clone detection approach in an enterprise software development environment to observe the unexplored characteristics of multi-language clone detection in software maintenance process.

References

[1] C. Kapser and M. W. Godfrey. ""Cloning Considered Harmful" Considered Harmful: patterns of cloning in software", J. Empirical Software Engineering, Vol. 13, 2008, pp. 645-692.

[2] C. J. Kapser and M. W. Godfrey. "Supporting the analysis of clones in software systems", J. Software Maintenance and Evolution: Research and Practice, Vol. 18, 2006, pp. 61–82.

[3] B. S. Baker, "On Finding Duplication and Near-Duplication in Large Software System", Proc. WCRE, 1995, pp. 86-95.

[4] M. Toomim, A. Begel, and S. L. Graham. "Managing duplicated code with linked editing", Proc. VLHCC, 2004, pp 173–180.

[5] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach", Science of Com. Prog., vol. 74, no. 7, 2009, pp. 470-495.

[6] K. Kontogiannis, P. Linos, and K. Wong, "Comprehension and maintenance of large-scale multi-language software applications," Proc. ICSM, 2006, pp. 497–500.

[7] iText. Website: http://itextpdf.com/, (Jul, 2012).

[8] S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo. "Comparison and Evaluation of Clone Detection Tools", IEEE TSE, Vol. 33 no. 9, 2007, pp. 577-591.

[9] I. Davis and M. Godfrey. "Clone detection by exploiting assembler", Proc. IWSC, 2010, pp. 77-78.

[10] B. S. Baker and U. Manber, “Deducing similarities in Java source from bytecodes”, Proc. ATEC, 1998, pp. 179-190.

[11] B. S. Baker. “On finding duplication and near-duplication in large software systems”, Proc. WCRE, 1995, pp. 86-95.

[12] N. Kraft, B. Bonds, and R. Smith. “Cross-language clone detection”, Proc. SEKE, 2008, pp 54 – 59.

[13] Mono. Website: http://www.mono-project.com/, (APR, 2012).

[14] ASXGUI. Website: http://sourceforge.NET/projects/asxgui/, (APR, 2012).

[15] C. Sadowski and G. Levin. “SimHash: Hash-based Similarity Detection”. Technical report, Google, December 2007.

[16] J. W. Hunt and T. G. Szymanski. “A fast algorithm for computing longest common cubsequences.” Communications of the ACM, Vol. 20, no. 5, 1977, pp. 350-353.

[17] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals", DAN, vol. 163, no. 4, pp. 845-848, 1965 (in Russian). English translation in Soviet Phys. - DokI., vol. 10, no. 8, pp. 707-710, 1966.

[18] S. Uddin, C.K. Roy, K.A. Schneider, and A. Hindle, “On the Effectiveness of Simhash for Detecting Near-Miss Clones in Large Scale Software Systems”, Proc. WCRE, 2011, pp. 13-22.

[19] C. K. Roy and J. R. Cordy. “NiCad: Accurate Detection of NearMiss Intentional Clones Using Flexible Pretty-Printing and Code Normalization", Proc. ICPC, 2008, pp. 172-18.

[20] M. S. Charikar, “Similarity estimation techniques from rounding algorithms”. Proc. STOC, 2002, pp. 380-388.

[21] iText.NET. Website: http://www.ujihara.jp/iTextdotNET/en/index.html, (APR 2012).

[22] H. Chernoff, “The use of faces to represent points in kdimensional space graphically”, Journal of the American Statistical Association, Vol. 68, no. 342, 1973, pp. 361-368.

[23] M. C. Chuah, S.G. Eick, “Glyphs for software visualization”, Proc. IWPC, 1997, pp. 183 – 191.

[24] J. W. Hunt and M. D. McIlroy. "An Algorithm for Differential File Comparison", Computing Science Technical Report, Bell Laboratories. 1975.

[25] I. J. Davis and M. W. Godfrey, “From Whence It Came: Detecting Source Code Clones by Analyzing Assembler”, Proc. WCRE, 2010, pp. 242–246.

[26] Soot Framework. Website: http://www.sable.mcgill.ca/soot, (APR 2012).

[27] G. M. K. Selim, K. C. Foo and Y. Zou. “Enhancing Source-Based Clone Detection Using Intermediate Representation”, Proc. WCRE, 2010, pp. 227-236.

[28] V. Juricic, “Detecting source code similarity using low-level language”, Proc. ITI, 2011, pp. 597-602.

[29] A. Santone, “Clone detection through process algebras and Java bytecode”, Proc. IWSC, 2011, pp. 73-74.

[30] The LLVM compiler infastructure. Website: http://www.llvm.org/, (APR, 2012).