Jurnal
Teknologi

Full Paper

ENHANCING A HYBRID PRE-PROCESSING AND TRANSFORMATION PROCESS FOR CODE CLONE DETECTION IN .NET APPLICATION

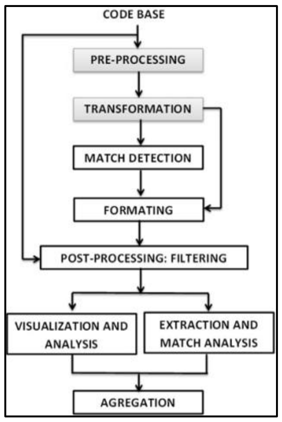
Article history
Received
2 February 2015
Received in revised form
8 October 2015
Accepted
12 October 2015

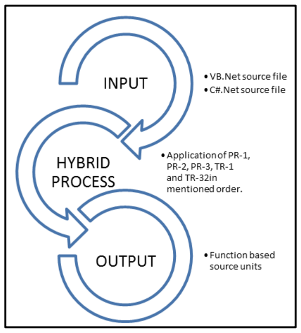
Al-Fahim Mubarak-Ali*, Shahida Sulaiman

Department of Software Engineering, Faculty of Computing, Universiti Teknologi Malaysia, 81310 UTM Johor Bahru, Johor, Malaysia

*Corresponding author
fahimbhai86@gmail.com

Graphical abstract


CODE BASE
PRE-PROCESSING
TRANSFORMATION
MATCH DETECTION
FORMATING
POST-PROCESSING: FILTERING
VISUALIZATION AND ANALYSIS
EXTRACTION AND MATCH ANALYSIS
AGREGATION


INPUT
• VB.Net source file
• C#.Net source file
HYBRID PROCESS
• Application of PR-1, PR-2, PR-3, TR-1 and TR-32in mentioned order.
OUTPUT
• Function based source units

Abstract

Pre-processing and transformation are the first two common processes that occur in a code clone detection process. The purpose of these two processes is to transform the source codes into a more representable form that can be used later on as input for code clone detection. Main issue arises in both of these processes is the application of the pre-processing and transformation rules might cause loss of critical information thus affecting the code clone detection results. Therefore, this work proposes a combination pre-processing and transformation process that can produce a better source unit representation of .Net platform source code which is C#. Net and VB.Net by enhancing an existing work that was done on Java language without affecting the critical information in the source code. The proposed enhancement was tested and the result showed that the proposed work was able to produce the expected source unit for the .Net platform languages together.

Keywords: Pre-processing process, transformation process, code clone

Abstrak

Pra-pemprosesan dan transformasi adalah dua proses pertama yang berlaku dalam proses pengesanan kod klon. Tujuan kedua-dua proses ini adalah untuk mengubah kod ke dalam bentuk yang lebih baik supaya ianya boleh digunakan seterusnya sebagai input untuk pengesanan kod klon. Isu utama yang timbul dalam kedua-dua proses ini adalah aplikasi teknik pra-pemprosesan dan transformasi yang mungkin akan menyebabkan kehilangan maklumat kritikal seterusnya menjejaskan keputusan pengesanan kod klon. Oleh itu, kajian ini mencadangkan satu proses gabungan antara pra-pemprosesan dan transformasi yang boleh menghasilkan kod unit yang lebih baik bagi kod dari platfom .Net iaitu C#.Net dan VB.Net dengan menambahbaik hasil penyelidikan sebelum ini yang menggunakan Java tanpa memberi kesan kepada maklumat yang kritikal dalam kod. Penambahbaikan yang dicadangkan ini telah diuji dan hasilnya menunjukkan bahawa kajian ini telah menghasilkan kod unit seperti yang diramalkan bagi kod dari platfom .Net.

Kata kunci: Proses pra-pemprosesan, proses transformasi, kod klon

1.0 INTRODUCTION

Code clone has been known to be an issue during maintenance of software. Code clone happens during software maintenance is due lack of awareness of newbie developers during maintenance of a software. A preliminary study in understanding problems in the code clone detection phase and modification phase among programmers especially novice programmers shows that most of the novice programmers may not be aware of the existence of code clones during the software maintenance phase [1].

Code clone is a common taxonomy used to refer codes that have been repeated multiple times in a program. Although code clone is a universal term used by researchers, yet there are also different terms used in addressing code clone. These difference occurs due to the different definition of similarity and associated level of tolerance allowed for the code clone [2].

The most commonly used terminology for code clone is categorized into four types which are Type 1, Type 2, Type 3 and Type 4. Type 1 is an exact copy of code without modifications with exception to white space and comments. Type 2 identifies identical copy syntactically. It only allows changes to variable, type or function identifiers. Type 3 is a copy code with further modifications. Modification involves statements that are changed, added, or removed. Type 4 is referred to clones that are modified syntactically [3] [4].

Apart from the mentioned category of code clones, other taxonomies used to refer code clone. Table 1 shows the summary of other code clone detection taxonomies.

Table 1 Taxonomy of clones

Taxonomy	Description
Structural clone	Structural clones are clones that highlight the similarities in design level (Hou *et al*., 2009). These clones reflect classes that are interrelated that come from design and analysis space at architecture level.
Functional clone	Functional clones are clones that occur at function or method level in software (Rattan *et al*., 2013).
Induced clone	Induced clones are clones that are purposely induced into a program for certain purposes such as for testing code clone detection approaches.
Temporal clone	Temporal clones are clones that occur from temporary during development when a program is executed.

The newbies tend to revise only defective codes they found first and not to search and revise their clones instead of looking for the clones in the whole software [1]. It is difficult for them to decide whether they should revise the files even if they are able to search for those files. Apart from that, it is difficult for a novice programmer to search for his target precisely with low cost. It is even more difficult to do it in large-scale legacy software because the target is vast and the terms are not standardized. Based on the outcome of the preliminary study done, it shows the newbie programmers have difficulty maintaining software due to the lack of knowledge and awareness regarding code clone. The practice that is adopted by them clearly does not reduce code clone but might even more code clones to occur. It is clear that it is hard for the newbie programmers to track and change code clones in large scale software systems due to non-standardization used in that large scale software.

Although code clone has been adopted during software development and maintenance, yet it has some residing disadvantages to a software system. Apart of being beneficial by speeding up development process [5]; and overcoming reuse mechanism and programming language [6, 7], the disadvantages of code clone in software development and maintenance includes:

1. Increase bugs and introduces new bugs in software
   If a code segment that contains a bug is reused by copy and paste technique without any changes, the bug of the original segment may remain in the pasted segments. Therefore, the probability for bug propagation increases in a system. Furthermore, new bugs might occur if the structure of duplicated code is reused without any changes.
2. Bad software design
   Due to the lack of good inheritance structure or abstraction, code cloning may cause bad design. Consequently, it makes the reuse of the inheritance or abstraction for future project implementation impossible thus badly affect maintainability of the software.
3. Halts system improvement
   Additional time and attention in understanding existing implemented code clone and concerns that need to be implemented. Therefore, it is difficult to add changes to the system.
4. Resource requirement escalates
   Since code clone increases the size of the program, hardware specification also needs to be upgraded. Furthermore, compilation time also has a detrimental effect on the edit-compile-test cycle as the compiler need to compile many codes in order to achieve the output.

In order to understand the process of code clone detection, Figure 1 shows a generic code clone detection process that is used by most researchers [8]. The generic process is through the general unification of existing steps and approaches that has been used by existing researchers. Most of the code clone detection tools adopt partially or fully the processes mentioned in the generic code clone detection process.

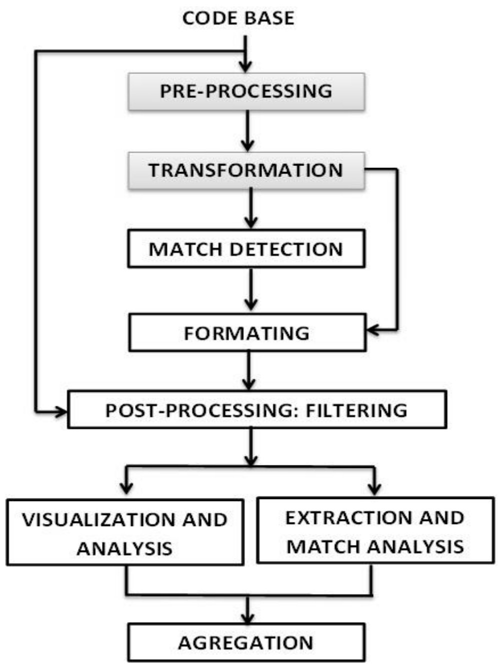

CODE BASE
PRE-PROCESSING
TRANSFORMATION
MATCH DETECTION
FORMATING
POST-PROCESSING: FILTERING
VISUALIZATION AND ANALYSIS
EXTRACTION AND MATCH ANALYSIS
AGREGATION

Figure 1 Generic Code Clone Detection Process [8]

Pre-processing is the first process in any code clone detection approach. The purpose of this process is to remove uninteresting parts, determine the source units and determine the comparison unit [8].

During the pre-processing process, all unwanted and uninteresting source code that is not for the purpose of comparison from is removed from this phase. The remaining source codes are then partitioned into source units. These source units are not in order; therefore it cannot be aggregated beyond the boundaries of the source units. There are several granularities for source unit such as classes, functions, methods, blocks and sequence of source code line. Based on the comparison function of a method, the source unit might be partitioned again into smaller units. These units might be divided into lines, or tokens for comparison purposes. The comparison units can also be derived from the syntactic structure of the source unit. Another important aspect of the source units is the order of the source units. The order of the source unit is important for comparison purposes.

Transformation is the second process in the generic process model and its main function is to transform the comparison units that were obtained from the previous process into another representation form that contains certain comparable properties. These comparable properties are attributed mainly to the match detection techniques that will be used for clone detection. Therefore, different transformation process yields different comparable properties. Table 2 shows the transformation approaches that can be used to extract the comparable properties.

Table 2 Transformation Approaches [8]

Transformation Approach	Description	Comparable Property
Tokenization	Each line of the source is divided Into tokens. These tokens correspond to a lexical rule of the involved programming language. The token lines are then formed into token sequences for the detection purposes.	Token or a group of tokens
Parsing	The entire source codes of the software are parsed into abstract syntax tree. The source unit and comparison units are represented in the form of sub tree.	Syntax or suffix tree
Normalizing identifiers	Usually applied in most of the approaches where the identifiers of the source code are replaced by a single token in such normalizations.	Token
Transformation of program elements	Apart from the normalization of the identifiers, several other transformation rules might be applied to the source code elements.	Depend on the applied rules

The issue arises in both of this processes is the use of pre-processing techniques and transformation rules might negligence of unneeded in the source codes such as package names and comments that influences the code clone detection results. The information might produce clones with different information; thus affecting the end result of code clone [2]. As an example, transformation of package names applied in CCFinder removes the initials of the package names in a Java source file. Package names and imports in Java applications are essential. Although it is changed by removing the initials, it doesn't serve as a clone in any clone granularity since clones are detected at a function manner. Another disadvantage of the existing works in Java is negligence in handling comments. Comment is an important component in source code. It serves as notes for developers and future developers when developing or maintaining software. The applied transformation rules often disregard the comments importance as it influences the clone detection processing time. Furthermore, token based source representations such as CCFinder [7] divides a source line into multiple tokens; thus requires longer processing time and larger memory.

Therefore, this work aims in enhancing the pre-processing and transformation process that reduces the loss information and processing time thus produce a better pre-processing and transformation process in improving code clone detection results for .Net platform programming language. These two processes are the main focus of this paper since the aforementioned problem can only be resolved in these two phases.

2.0 RELATED WORK

Most of the early works in code clone that uses text as intermediate representation are used in detecting clones in C source codes. A text to text source transformation was introduced for clone detection and change tracking [9]. This work uses this transformation to produces substrings that later on is used to detect code clones using substring match detection. The source transformation rules include in removing all white space characters, removing all white space except for line separators, replacing each sequence of white space characters by a single blank, removing comments, retaining only comments and replacing each identifier by an identifier marker. The source transformation is applied in various combinatorial manners.

A language independent approach was proposed to detect C language code clone [10]. This work uses source code that has gone through source transformation as the intermediate representation. Since it uses string based approach for detection purposes, the source transformation that is done on the code fragment is minimal so that it stays within the sight of string manipulation. A code fragment in this work is referred to a source code line. The transformation applied in this in removing comments and whitespaces until a condensed form of C code is obtained. A widely used token based code clone detection which is CCFinder [11] applies source transformation to its targeted Java and C# source files. Source transformation is the second process in code detection in this tool. The tokenized code goes through source transformation by the transformation rule and parameter replacement step. The transformation and parameter replacement applied for both Java and C# applications.

A hybrid technique of pre-processing and transformation process for code clone detection in Java language [12]. This work proposes a hybrid technique for pre-processing transformation process that transforms Java source code into source units based on a combination set of pre-processed and transformation rules. The proposed source units can be served as input for code clone detection techniques and approaches. The source units are composed in the hybrid form of text and tag. Table 3 shows the comparison of the related work with the proposed work.

Table 3 Comparison with Related Work

Feature	Representation Output	Language
Text to text [9]	String	C
Text to String [10]	Condensed code	C
CCFinder [11]	Tokens	Java, C++
Hybrid Technique [12]	String and tags	Java
Proposed Work	String and tags	C#.Net VB.Net

Although these works have successfully transforms the source code, yet there are still disadvantages to the previously applied transformation rules. The pre-processing and transformation is highly dependable on the programming language structure. Most of the programming language has a different way of addressing package names and imports. CCFinder [11] of package transformation is done by removing the initials of the package names in a Java source file. Package names and imports in Java applications are essential. Although it is changed by removing the initials, it doesn't serve as a clone in any clone granularity since clones are detected at a function manner.

Comment is also an important component in a source code file. Although there are rules applied in detecting, but it is important to know that the style of comment writing is different between programming languages. The hybrid technique done managed to remove the comments from the Java source file [12] but the applied transformation rules in other works often disregard the comments importance as it influences the clone detection processing time. Therefore, it is important to know style of commenting for each programming languages for the effectiveness of the applied rules. Token based source representations such as CCFinder [11] divides a source line into multiple tokens; thus requires longer processing time and larger memory.

3.0 THE PROPOSED WORK

This work adopts the hybrid technique of pre-processing and transformation process for code clone detection that was done in Java language [12]. This work enhances the hybrid technique by proposing pre-processing and transformation rules for the .Net platform language. The flow of this process is shown in Figure 2.

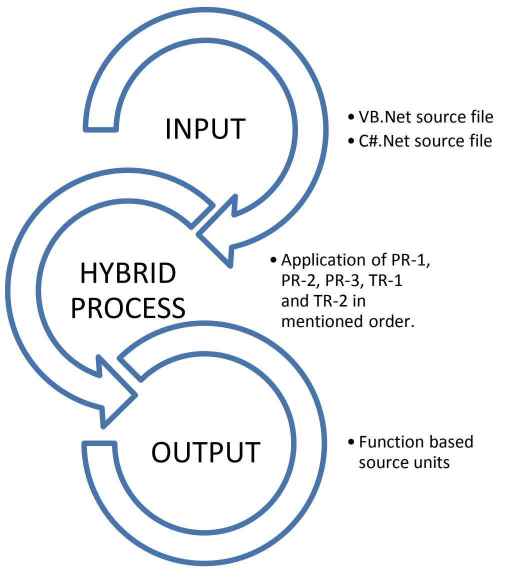

INPUT
• VB.Net source file
• C#.Net source file
HYBRID PROCESS
• Application of PR-1, PR-2, PR-3, TR-1 and TR-2 in mentioned order.
OUTPUT
• Function based source units

Figure 2 The flow of the proposed work

The input used for this technique is C#.Net and VB.Net applications. .Net platform language applications include .Net source library, libraries and extended libraries. Therefore, C#.Net and VB.Net source files (files that contain the file extension of .cs and .vb) are extracted out from the application.

The improved hybrid technique consists of combination of five rules which are PR-1, PR-2, PR-3, TR-1 and TR-2. The purpose of these rules is to process and transform the source files into source units without losing too much information of the source codes in the source files. This rules are also to generalize the source codes so that more variation in the code clone detection result and analysis [4]. The rules applied influence the code clone detection results. Therefore, the rules must not be too rigid so that the information in the source codes can be sustained for clone detection purposes. Furthermore, this rules are also designed to overcome the aforementioned gaps. The rules applied for the .Net platform language are described in Table 4.

The output is function based source units that are obtained after going through the hybrid technique. The representation form of this source units are hybrid of text and token that are stored in a single .Net source file.

If,

nF = .Net folder of tested application;
F = files;
cSF = C#.Net source file;
vSF = VB.Net source file;
cSCL = C#.Net source code line;
vSCL = VB.Net source code line;
PR-1 = First pre-processing rule;
PR-2 = Second pre-processing rule;
PR-3 = Third pre-pocessing rule;
TR-1 = First transformation rule;
TR-2 = Second transformation rule;

Table 4 The Hybrid Rules [12]

Rule#	Description
PR-1: Remove package and import statements	This rule is designed to remove the import statements and package names from the source file.
PR-2: Remove comments	This rule attempts to remove comment lines occur in new lines.
PR-3: Remove empty lines	This rule is to remove all empty lines in the source file.
TR-1:Keywords regularization with identifiers	Keywords are words that have a predefined meaning in a programming language. The keywords that are replaced with unique identifiers in this rule are: • string -> [s] • char -> [c] • int -> [i]
TR-2: Regularize function access to public	This rule regularizes all the function accesses into a single function access; which is public.

.

Therefore, the pseudocode of the hybrid technique for the improved hybrid technique of pre-processing and transformation process for the .Net platform is:

```
1 Read nF
2   if nF is empty
3      Read next nF
4   else if nF is not empty
5     Read F in cSF and vSF
6       if F is not cSF or vSF
7          Remove F
8       else if F is cSF and vSF
9          for each cSF and vSF
10            Read cSCL and vSCL
11              if cSCL and vSCL is empty
12                 Continue to next cSCL and vSCL
13              else if cSCL and vSCL is not empty
14                 Apply PR-1
15                 Apply PR-2
16                 Apply PR-3
17                 Apply TR-1
18                 Apply TR-2
19 Continue from 9 to 19 for all cSF and vSF
```

4.0 RESULT

4.1 Experimental Setup

The proposed work was developed and tested using Netbeans 8.0. The test used a workstation with the specification of 3.20GHz CPU, 12GB of memory with

Windows 8.1 as its operating system. Dataset for evaluation of proposed work is a challenge as current benchmark dataset available is from Bellon's benchmark data [2] that consist of C and Java only. A total of six applications were used to test the workability of the proposed work. Three of the application files were C#.Net project files and the other three application files were VB.Net project files from open source code and project repository [13, 14]. Table 5 and 6 shows the details of each project files respectively.

Table 5 C# .Net dataset

Project	Language	Source File	Folder Size (MB)
Satsuma 0.1alpha [14]	C#.Net	62	0.5
NClass v2.04 [14]	C#.Net	540	3.7
SharpDevelop 5.1.0.4936 [14]	C#.Net	11515	71.9

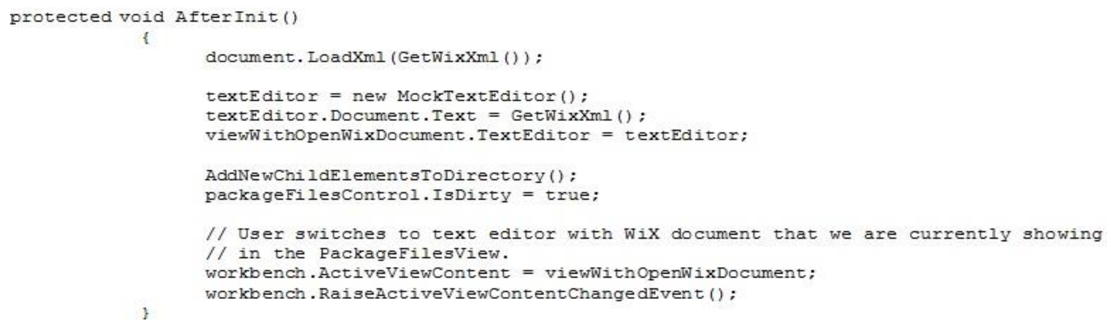
Table 6 VB .Net dataset

Human Resource Management System [13]	VB.Net	70	8.05
Hotel Management System [13]	VB.Net	99	4.16
Medical Information System [13]	VB.Net	133	2.18

The result obtained is evaluated on two aspects which are the representation of the source units and the runtime performance of this process.

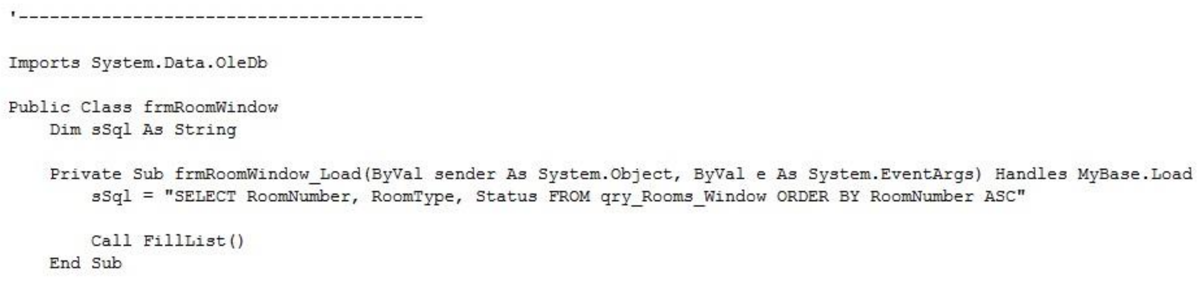
4.2 Source Units

Figure 3 shows the sample C#.Net source code while Figure 4 shows the sample VB.Net source code taken from the data set. The C#.Net source code in Figure 3 contains a function that starts with 'protected', empty lines and comments between the line of codes. The sample VB.Net source code shown in Figure 4 contains package name, comments, empty lines and keyword of 'string'. Both of the sample codes in Figure 3 and Figure 4 are common view of source code in a .Net platform language.



```
protected void AfterInit()
        {
                document.LoadXml(GetWixXml());

                textEditor = new MockTextEditor();
                textEditor.Document.Text = GetWixXml();
                viewWithOpenWixDocument.TextEditor = textEditor;

                AddNewChildElementsToDirectory();
                packageFilesControl.IsDirty = true;

                // User switches to text editor with WiX document that we are currently showing
                // in the PackageFilesView.
                workbench.ActiveViewContent = viewWithOpenWixDocument;
                workbench.RaiseActiveViewContentChangedEvent();
        }
```

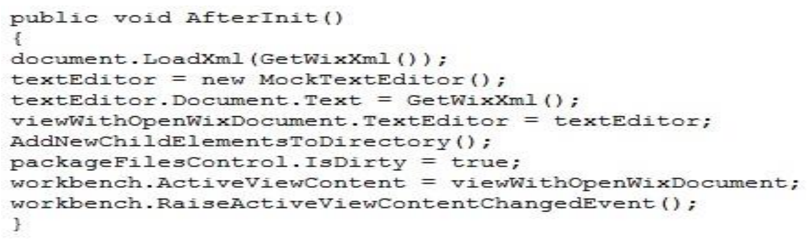
Figure 3 C#.Net sample source code



```
'---------------------------------------

Imports System.Data.OleDb

Public Class frmRoomWindow
    Dim sSql As String

    Private Sub frmRoomWindow_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
        sSql = "SELECT RoomNumber, RoomType, Status FROM qry_Rooms_Window ORDER BY RoomNumber ASC"

        Call FillList()
    End Sub
```

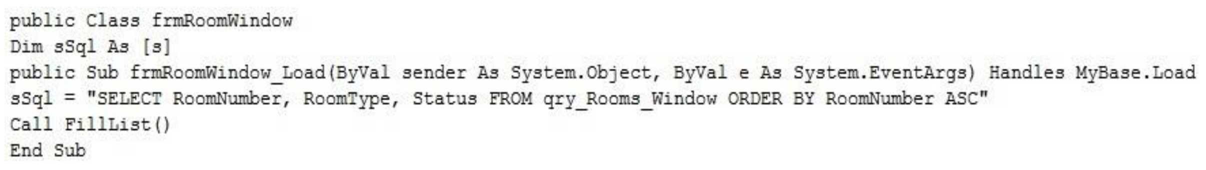
Figure 4 VB.Net sample source code

Figure 5 shows the application result of the improved hybrid technique on C#.Net while Figure 6 shows the application result of the hybrid technique on VB.Net. The package names, comments and empty lines from both of the C#.Net and VB.Net source code is gone due the application of the rules. The keyword in Figure 4 has also been changed according to the applied rules.



```
public void AfterInit()
{
document.LoadXml(GetWixXml());
textEditor = new MockTextEditor();
textEditor.Document.Text = GetWixXml();
viewWithOpenWixDocument.TextEditor = textEditor;
AddNewChildElementsToDirectory();
packageFilesControl.IsDirty = true;
workbench.ActiveViewContent = viewWithOpenWixDocument;
workbench.RaiseActiveViewContentChangedEvent();
}
```

Figure 5 C#.Net sample source code output



```
public Class frmRoomWindow
Dim sSql As [s]
public Sub frmRoomWindow_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
sSql = "SELECT RoomNumber, RoomType, Status FROM qry_Rooms_Window ORDER BY RoomNumber ASC"
Call FillList()
End Sub
```

Figure 6 VB.Net sample source code output

4.3 Runtime Performance

Runtime performance refers to the overall time taken to complete the process. Figure 7 shows the runtime performance taken by the proposed work for C#.Net source code applications.

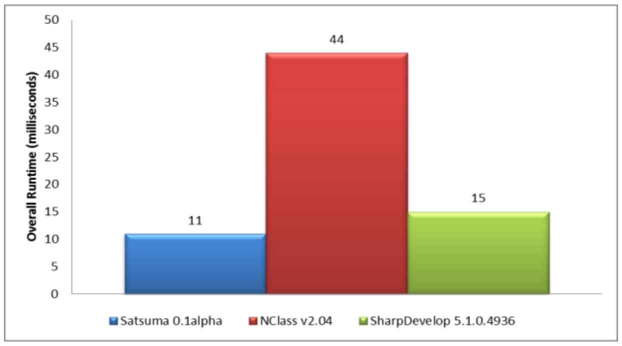


Figure 7 Runtime performance for C#.Net application

Satsuma 0.1alpha has the lowest runtime performance which is 11 milliseconds compared to SharpDevelop 5.1.0.4936 which has 15 milliseconds. NClass v2.04 has the highest runtime performance with 44 milliseconds.

Figure 8 shows the runtime performance taken by the proposed work for VB.Net source code applications.

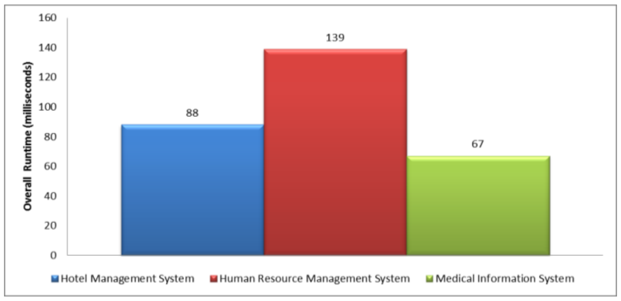


Figure 8 Runtime performance for VB.Net application

Medical Information System has the lowest runtime performance compared to other VB.Net applications with 67 milliseconds compared to Hotel Management System which has 88 milliseconds. Human Resource Management System has the highest runtime performance with 139 milliseconds.

5.0 DISCUSSION

Based on the experiments and results, the proposed work was able to produce function based source unit from the enhancement of the hybrid process of pre-processing and transformation. The proposed work which was enhanced from a previous work [12], currently able to produce a better .Net platform languages source representation for code clone detection purposes. Although there are improvements based on the experimental results, yet there are issues that can deter the validity of the results.

Sample data used for experiments are three C#.Net and three VB.Net applications. The size and structure of the source code for these six applications are not same and vary each other. Each application has different amount of line of codes (LOC) and source files. The results might vary with more sample data with a bigger amount of the LOC and source files. The lack of standardisation of data set is one of the biggest threats to the validity of the data.

Furthermore, the naming convention, code structure, amount of functions, system architecture and coding styles vary between these sample applications. Since it is vary between the applications, the runtime performance is affected by these variants.

Another rising issue to validity of the results is the hardware specification. Hardware specification that is used for this experiment support current sample data. As the technology improves and updates, the improved hardware specification will result in better runtime performance.

6.0 CONCLUSION

This paper explains the proposed work which is the enhancement of hybrid process of pre-processing and transformation with an aim to produce better source units for C#.Net and VB.Net source codes without jeopardizing the information of a source file. The experimental results show that the hybrid process managed to produces expected source units.
As for future work, the hybrid process will be refined to support other structural and procedural programming language detect code clone for the purpose of code clone detection analysis.

Acknowledgement

The authors acknowledge Fundamental Research Grant Scheme (FGRS) under the Ministry of Education, Cost Centre 4F263 that partially supports this work.

References

[1] Kawaguchi, S., Yamashina, T., Uwano, H., Fushida, K., Kamei, Y., Nagura, M., *et al.* 2009. SHINOBI: A Tool for Automatic Code Clone Detection in the IDE. Presented at the Reverse Engineering, 2009. WCRE '09. *16th Working Conference on, 2009.*

[2] Bellon, S., Koschke, R., Antoniol, G., Krinke, J., and Merlo, E. 2007. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*. 33: 577-591,

[3] Roy, C. K. 2009. Detection and analysis of near-miss software clones. In *IEEE International Conference on Software Maintenance*. 447-450.

[4] Mubarak-Ali A.-F., Syed-Mohamed S.-M., and Sulaiman S. 2011. An Enhanced Generic Pipeline Model for Code Clone Detection. In *5th Malaysian Conference in Software Engineering (MySEC)*. 434-438.

[5] Hou, D., Jacob F., and Jablonski P. 2009. Exploring the Design Space of Proactive Tool Support for Copy-and-Paste Programming. Presented at the *Proceedings of the 2009 conference of the Centre for Advanced Studies on Collaborative Research (CASCON '09), Toronto, Ontario, Canada.*

[6] Kapser, C. J. and Godfrey, M. W. 2006. Supporting the Analysis of Clones in Software Systems. *Journal of Software Maintenance and Evolution: Research and Practice*. 18: 61-82.

[7] Kapser, C. J. and Godfrey, M. W. 2008. Cloning Considered Harmful Considered Harmful: Patterns of Cloning In Software. *Empirical Software Engineering.* 13: 645-692,

[8] Roy, C. K. and Cordy, J. R. 2007. A Survey on Software Clone Detection Research. SCHOOL OF COMPUTING TR 2007-541. QUEEN'S UNIVERSITY. 115.

[9] Johnson, J. H. 1994. Substring Matching for Clone Detection and Change Tracking. In *International Conference on Software Maintenance*. 120-126.

[10] Ducasse, S., Rieger, M., and Demeyer, S. 1999. A Language Independent Approach for Detecting Duplicated Code. Presented at the *Proceedings of the IEEE International Conference on Software Maintenance*.

[11] Kamiya, T., Kusumoto, S., and Inoue, K. 2002. CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering*. 28: 654-670.

[12] Mubarak-Ali A.-F. and Sulaiman S. 2014. A Hybrid Technique in Pre-Processing and Transformation Process for Code Clone Detection. In *Software Engineering Conference (MySEC), 2014 8th Malaysian*. 102-107.

[13] FreeStudentsProjects. 2014. FreeStudents Projects. Available: http://www.freestudentprojects.com/.

[14] Sourceforge.net. 2014. Sourceforge.net. Available: http://sourceforge.net/home.html.