International Journal of Computer Information Systems and Industrial Management Applications.
ISSN 2150-7988 Volume 5 (2013) pp. 564-570

Code Clones in Program Test Sequence Identification

Anupama Surendran[1], Philip Samuel[2] and K. Poulose Jacob[3]

[1] Department of Computer Science, Cochin University of Science &Technology, Kochi-22, Kerala, India
anupama.deepak@gmail.com
[2] Information Technology, School of Engineering, Cochin University of Science & Technology, Kochi-22, Kerala, India
philips@cusat.ac.in
[3] Department of Computer Science, Cochin University of Science &Technology, Kochi-22, Kerala, India
kpj@cusat.ac.in

***Abstract*: Code clones are portions of source code which are similar to the original program code. The presence of code clones is considered as a bad feature of software as the maintenance of software becomes difficult due to the presence of code clones. Methods for code clone detection have gained immense significance in the last few years as they play a significant role in engineering applications such as analysis of program code, program understanding, plagiarism detection, error detection, code compaction and many more similar tasks. Despite of all these facts, several features of code clones if properly utilized can make software development process easier. In this work, we have pointed out such a feature of code clones which highlight the relevance of code clones in test sequence identification. Here program slicing is used in code clone detection. In addition, a classification of code clones is presented and the benefit of using program slicing in code clone detection is also mentioned in this work.**

***Keywords*:** Code clone, program slicing, test sequence, static slicing

I. Introduction

Generally there is a misconception that software testing is performed as the final step of software development life cycle. As software testing plays an inevitable role in software development process, applying software testing only in the final stage of software development will make the whole process more complicated. This is due to the fact that the programmer sometimes will have to go back to the initial stage of software development to track the error. Due to the time constraints meet by the software developers in developing the software for specific applications, the process of automating software testing has gained so much importance. Therefore the software should be properly tested to check whether the developed software satisfies the user requirements before handing over the final version to the users. Due to these strict constraints meet by the programmers, some effective methods should be adopted to make the whole software testing process easier and in this work we have showed how the reusable property of software source code is utilized in software testing process. Several research works shows that about 7% to 23% of the software source code is cloned and in some cases nearly 50% of the source code is cloned [1]. Software reuse is the process of using existing software components rather than building from the scratch and the concept of code clones is derived from this idea [1]. A code clone can be defined as a set of program statement which may be contiguous or non-contiguous and which repeats in several other parts of the same program or in different parts of the same program or in different files of the same application program [4]. Even though code reuse saves time and manual effort, some researchers claim that software code clone increases the software maintenance cost. For example, if a programmer makes any slight modification in a code clone, and if the same change is not made in the other code clones present in the program, it may cause inconsistency. Similarly when the code clones are created, the programmer sometimes forgets to map the variable values. In such cases the code clones will not be harmonic and this causes errors in the program. These are some of the problems which are to be faced in code clone maintenance. Due to these issues related with the maintenance of software code clones, proper detection of code clones in each and every stage of software development is essential. Utilizing the positive aspects of code clones in an appropriate way can result in marked changes in the field of software testing industry. Many of our day to day computer applications take advantage of the code reuse property. The main reason behind this code reuse mentality is to make the software development process easier rather than writing the source code from scratch. For example, while making a newer version of operating system, developers are not writing the program code from the scratch, rather they try to concentrate only on the new functions which are to be integrated into the newer version. Similarly several libraries of mathematical routines are reused instead of developing them each and every time. The main point to be noted here is that the code developers can concentrate in developing the new features of the software rather than putting effort on the old problems repeatedly. This has the potential benefit of reducing the source code development time and cost. In testing, the need of using code clones in test sequence generation can be applied to such scenarios as there is no need to check or identify test sequence repeatedly for similar clones. This reduces the program

tester's effort as they have to consider only the minor variations made in the code clones. Apart from testing, code clones can be used in many other applications such as program comprehension, program compaction, checking plagiarism etc.

In this paragraph, a brief overview of the significance code clones in real world application is given. Code plagiarism is the repetition of source code in a program and detection of code clones helps to eliminate code plagiarism. There are several approaches to detect plagiarism. In string based approach the exact matches are obtained and in token based approach the source program is converted to tokens to detect the similarity between token sequences. Another approach is parse tree based approach in which a parse tree of the source program is constructed to detect code similarity. In Program dependence based approach, a program dependence graph is constructed to detect the code clones at a higher level and in metrics based approach, code segments are assigned certain scores based on some metrics and using this metrics code clones are detected. Program comprehension is the process of analyzing the program code for software maintenance purpose. Identification of similar code in a program makes program comprehension easier as it eliminates the need of repeated analysis of similar source code. Program comprehension methods include text based, lexical, syntactic, graphical and execution based methods. Program compaction is the process of reducing the code size. Code size can be reduced by detecting code clones present in a program and replacing the common code by code refactoring technique. In code refactoring, similar code from a program block is moved to a node which succeeds or precedes the block. The repeated portions of the source code will be placed in abstract procedures and a call to these abstract procedures is made instead of using the repeated code sequences [14] This paper describes a program slicing based approach for code clone detection and their use in test sequence identification. The rest of this paper is organized in the following manner. Section II gives the most related work and section III gives the reasons for the existence of code clones in software system and some of the basic terminologies. Section IV gives the basics of slicing and section V illustrates how code clones are detected generated using slicing and control flow graph. This section also explains the significance of code clones in program testing and the importance of this work. Section VI gives the conclusion and future modifications which can be applied to this existing work.

II. Related work

There are excellent research works going in the field of code clones and utilization of clones for various applications. To the best of our knowledge, no works are reported in the field of using code clones for software test sequence generation. C. K. Roy and J. R. Cordy [2007] have done an excellent survey on software code clones and their detection methods. R. Komondoor and S. Horwitz [2001] has explained how program slicing techniques are used program dependence graphs can be used to identify the code clones in a program. Similarly J. Krinke [2001] has also explained how source code clones are identified from program dependence graphs. Compared to these two approaches our method of code clone detection using slicing is more variable specific and we have used a slicing approach which gives all the dependency present in the program with less effort. In this work we have mainly concentrated on code clone detection part and the

III. Basic Terminologies

As pointed out in the Section I, code reuse is a common practice in software industry and the extent of code reuse varies in different industry. As evidence to this fact, we have taken the example of Hewlett- Packard software industry, where three levels of code are considered in software developed. They are new code, reuse code and leveraged code. In reuse code, the existing code is used without making any modification and in leveraged code some modifications is made to the original source code. Figure 1 shows an evidence of reuse on six different firmware projects at Hewlett-Packard. The following subsections describe some of the common terminologies related to code clones.

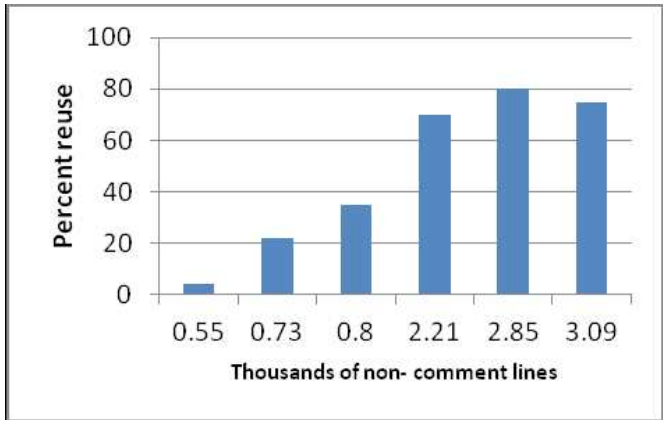

Figure 1. Graph illustrating the percentage of code reuse in HP industry

A. Reasons for existence of clones in software systems

Code clones exit in a software system due to a number of reasons. In software industry we have already seen that code reuse is a common practice and during the code reuse, code clones are mainly generated due to the copy/paste practice. Similarly, a hardware driver can be reused in a similar hardware platform. The code clones can be introduced due to programming approaches also. For example, when two similar software systems are combined to produce a new system, there is a chance of existence of code clones in the newly developed system, even if they are developed by different personal. In generative programming, code clones exit due to the reuse of similar templates which contains the same logic. During software maintenance process, the developers will have to update existing software. Here the developer reuses the existing software code and tries to incorporate the modifications alone. In programming languages, if properties such as inheritance, parameter passing etc. is not present, then the programmer will have to rewrite the whole source code. If the programmer is not able to understand the functions of a software system, then they will be force to reuse the existing code to develop a new one. Similarly if a programmer has to complete a project within a deadline, then they may copy parts

of source code and reuse them in other areas in order to save time.

B. Code Fragment

A group of program statements or lines within a program is known as code fragment. This set of sentences can be either conditional sequence set or it can be well defined functions or simply a group of statements.

C. Definition (Code Clones)

Consider a set of program statements 'CLN'. Another set of program statements 'CLNc' can be considered as a code clone of 'CLN', if they contain the same set of program statements or if they have some similar properties up to certain extend. In order to check the degree of similarity which exists between code fragments 'CLN' and 'CLNc', we have to categorize the clones according to their behavioral properties [5].

D. Categories of code clones

There are mainly four types of code clones. They are type1, type 2, type 3 and type 4 clones. Each of these is explained below.

1) Copy Clones

This type of clones is included in type 1 category. In this type of code clones, the code fragments will be exactly the same. The code clone formed will be the exact copy of the original code fragment. There will be some minor changes in white spaces, comments etc.

Code1:-	Code 2:-

int s=0, p=1, n=5;
while (n > 0)
{
s= s+n;
p=p* n;
n= n-1;
}
	int s=0, p=1, n=5;
	while (n > 0)
	{
	s= s+n;
	p=p* n;
	n= n-1;
	}

Table 1. Copy Clones

2) Renamed Clones

This type of clones will come under type 2 category. In this type, the code clone formed will be syntactically similar to the original code. There will be changes in the name of identifiers, functions, literals etc. along with minor variations in white spaces, comments etc. Here we can see that code 2 is similar to code 1. The main difference is the change in the variable names. Therefore these two types of clones can be included in type 2 category

Code1:-	Code 2:-

int s=0, p=1, n=5;
while (n > 0)
{
s= s+n;
p=p* n;
n= n-1;
}
	int a=0, b=1, c=5;
	while (c > 0)
	{
	a=a+c;
	b=b*c;
	c=c-1;
	}

Table 2. Renamed Clones

3) Modified Clones

This type of code clones comes under type 3 category and they can be considered as a modification of type 2 clones. They will have all the features of type 2 with some additional features. Additional features means that some lines will be changed, deleted or added to the clone. We can see that Code 2 is an example of Type 2 clone. Comparing Code 2 and Code 3, we can see that code 3 is having all the features of code 2 and apart from that, there is one extra statement present in code 3. Therefore these two types of clones can be included in type 3 category.

Code 2:-	Code 3:-

int a=0, b=1, c=5;
while (c > 0)
{
a=a+c;
b=b*c;
c=c-1;
}
	int a=0, b=1, c=5,s=0;
	while (c > 0)
	{
	a=a+c;
	b=b*c;
	s= a+b;
	c=c-1;
	}

Table 3. Modified Clones

4) Interpreted Clones

This type of clones will have entirely different syntax and they come under type 4 category. There will be no similarity between the program text or program lines. Even though they differ syntactically, their functionality remains the same. Here code 1 and code 2 are syntactically not similar, but their behavior is the same. Therefore these two can be considered as Type 4 clones.

Code 1:-	Code 2:-

int p, q, r;
r=0;
while (p>0)
{
r=r+y;
p=p-1;
}
	int s, t, u;
	u=0;
	while (s<0)
	{
	u= u-y;
	s=s+1;
	}

Table 4. Interpreted Clones

IV. Overview of Slicing

We have used program slicing technique to identify the code clones in a program. Code clones are to be detected from the program source code. It may not be always practical to check the whole program which may contain thousands of lines of code to find the presence of code clones. In several situations, the program tester will be interested only in particular parts or function of the source program which is supposed to perform certain important tasks. In such a scenario, it is not advisable to analyse the program lines one by one as this will only cause unnecessary waste of time and effort. Program slicing is applied in such situations. Instead of analysing the whole program, slicing converges the focus to some specific program parts implied by the slicing criterion [12]. It was Weiser who introduced program slicing in 1979 and his work encouraged many applicative research works in this field.

The structural similarity of the control flow graphs which are constructed for the various slices in the program are used to identify the code clones. Sliced statements give the variable

dependencies present in the program and also eliminate the need for unnecessary checking of the whole program.

A. Slicing

Process of reducing the complexity of a program by removing the irrelevant statements from a program is known as program slicing or in other words, a slice is a miniature version of the source program. According to Weiser, a static slice is a set of statements that directly or indirectly affect the value of a variable at a given program point and this point is known as slicing criterion. The slicing criterion is denoted by (S, V) where 'S' is the statement or line number and 'V' is the variable in the program. The static slices were uncertain in nature. Therefore B. Korel and Laski [1988] introduced the concept of dynamic slicing to target the programmer's attention only to the relevant parts of the program. Runtime information about a program is used in dynamic slices.

Some statements of the program will have an effect on the values computed at some point of interest. The point of interest is known as slicing criteria and it is represent as C= (x, y) where x is the statement present in a given program and y is the subset of variables present in the program. A static slice constructed by ignoring those parts of the program that are not relevant to the values stored in the chosen set of variables at the chosen point specified in the slicing criterion. Given a variable 'v' and a point of interest 'n', slice will be constructed for v at n.

The set of statements that affects the value of a variable for one specific input is known as dynamic slice. In dynamic slicing we have to consider three parameters. First one is the point of interest within the program, second one is the variable and the third one is the sequence of input values for which the program was executed. Dynamic slicing criterion is defined as C= (x, y, i). Here 'x' is the statement in the program, 'y' is the subset of variables in the program and 'i' is the input value [10].

Code Fragment:-	Slice obtained for Slicing Criterion:- (p, 9)
1 scanf("%d",&n); 2 s=0; 3 p=0; 4 while (n>0) 5 { 6 s=s+n; 7 p=p*n; 8 n=n−1; 9 }	1 scanf("%d",&n); 3 p=0; 4 while (n>0) 5 { 7 p=p*n; 8 n=n−1; 9

Table 5. Static slice example

Code Fragment:-	Dynamic slice for criterion:- (p, 10, n=0)
1 scanf("%d",&n); 2 s=0; 3 p=0; 4 while (n>0) 5 { 6 s=s+n; 7 p=p*n; 8 n=n−1; 9 } 10 printf("%d %d",p,s);	3 p=0

Table 6. Dynamic slice example

Backward slices give all the program statements which affect the value of a particular variable at a particular point [6, 7]. Backward slicing criteria is defined as C=(x, y). Here 'x' is the statement number and 'y' is the slice variable

Forward slices give all the program statements which are affected by declaring a variable at a given point in the program [6, 7, 9]. Forward slicing criteria is defined as C=(x, y). Here 'x' is the statement number and 'y' is the slice variable. After applying program slicing technique, CFG of the slices [11] should be created. In our clone detection approach we are applying static slicing technique for clone detection. The structural similarity present in the control flow graph is taken in to consideration to detect the code clones. These are clearly illustrated in the next section.

V. Identifying Test Sequences using Code Clones

In the above sections, we saw the relevance of code clones and have explained the different types of code clones. This section explains our idea of utilizing code clones for identifying similar test sequences during program testing [13]. There is no need to identify separate test sequence for similar code clones and thus the testing effort is reduced. Here, program slicing is used to detect the presence of code clones [2, 3]. Consider the following example in Figure 2.

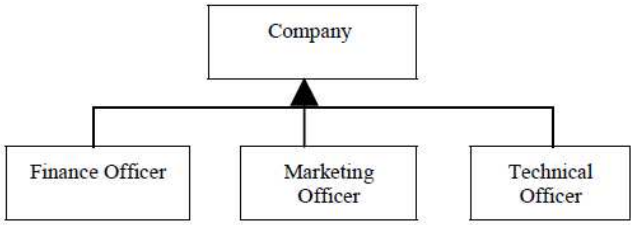

Figure 2. Employee Classification

Here we can see that there are three classes of employee under the Company class. They are Finance officer, Marketing officer and Technical officer. The salary calculation code of Finance officer, Marketing officer and Technical officer is as follows:-

Finance Officer	Marketing Officer	Technical Officer

int basic, da, hra, total;
basic= 1000;
if(basic >= 1000)
{
da=100;
hra=50;
}
else
{
da=50;
hra=25;
}
total=basic+da+hra;
	int basic, da, hra, total;
	basic= 1000;
	if(basic >= 1000)
	{
	da=100;
	hra=50;
	}
	else
	{
	da=50;
	hra=25;
	}
	total=basic+da+hra;
		int basic, da, hra, total;
		basic= 2000;
		if(basic >= 1000)
		{
		da=100;
		hra=50;
		}
		else
		{
		da=50;
		hra=25;
		}
		total=basic+da+hra;

Table 7. Salary calculation code of different officer classes

Code clones exist in the above three categories of officers. Checking the Finance and Marketing officer class, we can notice that there is no change in the ' total salary ' calculation steps. The salary calculation method, the basic salary value, hra value and da value are exactly same for both

the classes. Therefore these set of program statements can be considered as code clones. In this scenario, if we identify the test sequence statements for the Finance officer class, then there is no need to identify the test sequence again for the Marketing officer class, as we have already detected the test sequence for its clone. For this, the first step is that the tester should be able to detect the presence of code clones in Finance officer and Marketing officer class.

For detecting the code clones, program slicing principle is initially applied to various classes. Then we are constructing the control flow graph of the slices obtained. We are mainly considering the structural similarity present in the CGF to identify the code clones. We are using slicing as the initial step of clone detection in order to avoid unnecessary checking of the whole program. Sliced statements will give an overview of the variable dependency present in the program. Consider the Finance Officer class in the *Table 7* given above. Initially we perform static slicing with respect to the variable 'total' present in the statement 'total= basic + da+ hra'. The resultant statements obtained is named as Set I. Next we perform static slicing in Marketing officer class. Here also, static slicing is done with the variable 'total' present in the statement 'total = basic + da + hra'. This is named as Set II. Set I and Set II is given in *Table 8* given below.

Set I	Set II

1 basic= 1000;
2 if(basic >= 1000)
3 {
4 da=100;
5 hra=50;
6 }
7 else
8 {
9 da=50;
10 hra=25;
11 }
	1 basic= 1000;
	2 if(basic >= 1000)
	3 {
	4 da=100;
	5 hra=50;
	6 }
	7 else
	8 {
	9 da=50;
	10 hra=25;
	11 }

*Table 8.*Slice of Marketing officer and Finance officer class

The next step is to draw the control graph of the static slices obtained for both the classes. Here it can be noticed that the control flow graphs obtained for the static slice Set I and Set II are the same.

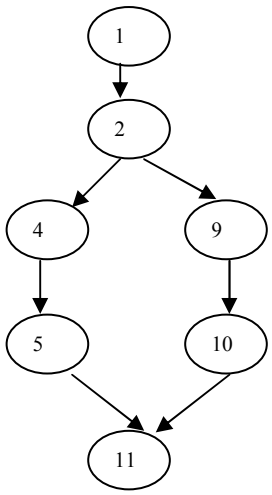

Figure 3. CFG of Set I

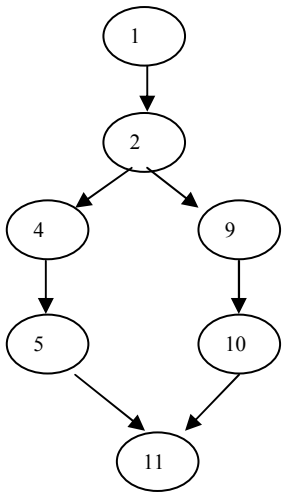

Figure 4. CFG of Set II

We are making use of the structural similarity present in the control flow graph to detect code clones. For structural similarity, the CFG of Set I and Set II are checked to verify whether they are isomorphic graphs or not. Two graphs, G= {V, E} and G1= {V, E} are said to be isomorphic graphs if there exists one- to- one correspondence between their vertices and between their edges such that the incidence relationship is preserved. Suppose that an edge ' e' has end vertices V1 and V2 in G, then the corresponding edge ' e' in ' G1' must be incident on vertices V11 and V12 that correspond to V1 and V2 respectively. Here both the control flow graphs satisfy these properties. Here the graphs of Set I and Set II have the same number of vertices, same number of edges and same degree sequence. Therefore these two control graphs can be regarded as code clones

After detecting the code clones from the control flow graph, the next step is to check the node content of each graph. We are checking each and every node of the two code clones. Here all the nodes of both the graphs are equal. This indicates that there is no need to identify test sequence separately for the clones. If the test sequence for one of the clones is generated then the same test sequence is applicable to the other clone also. Substituting appropriate values in the test sequence statements of code clones simplifies program testing process. In the example given above, the test sequence for both the Finance officer class and Marketing officer class is identified from the code clones. The test sequence identified is given below.

Test Sequence:-

```
basic= 1000;
if (basic >= 1000)
{
da=100;
hra=50;
}
 else
{
da=50;
hra=25;
}
total=basic+da+hra;
```

Therefore, the total salary of both Finance officer class and Marketing officer will be 'total= 1150' which is got by substituting the desired values in the test sequence generated.

A. Need for interpreting control flow graph node contents

Even though the above given example in section V correctly identifies the test sequences from code clones, interpreting the control flow graph node contents is required in certain situations. This is required for identifying the correct test sequence and to generate correct test data values from them. This can be explained with the help of the same example given Figure 1. Already we have found out that code clones exist in Finance officer and Marketing officer class. Now consider the Technical officer class. Apply static slicing in Technical officer class with respect to the variable ' total' present in the statement ' total = basic + da + hra. The result will be a set

of statements given in Set III. The CFG of Set III is given in Figure 5.

Set III:-
```
1 basic= 2000;
2 if(basic >= 1000)
3 {
4 da=100;
5 hra=50;
6 }
7 else
8
9  da=50;
10 hra=25;
11}
```

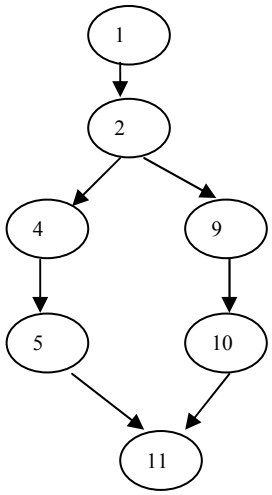

Figure 5. CFG of Set III

After constructing the control flow graph for Set III, we have to compare the CFG of Set I, Set II and Set III. We can see that all the three control flow graphs are structurally similar. The next step is to check the contents of each and every node of all the three control flow graphs. In the above section itself we have already found that the CFG of Set I and Set II are similar. In the CFG of Set III the contents of the first node is different from CFG contents of Set I and Set II. In Set I and Set II, the first node is having the content as' basic= 1000'. The CFG of Set III is having the content as' basic=2000'. Therefore there is a need for mapping of statements here. Even though the test sequence is almost similar in Set I, Set II and Set III, there is a minor difference in Set III CFG node values. The program tester should identify these variations in node values so that correct test data value is generated from the test sequences.

B. Significance of code clones in program testing

In our method, we have made use of code clones for test sequence identification. Although from the maintenance point of view, code clones are generally considered as one of the most challenging problems faced in software development process, we have utilized some of the strong features of code clones that make them applicable in test sequence identification. Initially, we have applied program slicing to the source program. This is done in order to avoid unnecessary checking of the whole source program. Apart from that, slices obtained will give the dependence information which exists between the variables present in the program. In some cases, checking the CFG alone will not be enough to identify the code clones and this flaw in the detecting code clones will cause errors in test data generation process. This is due to the fact that the structural resemblance present in the CFG alone may not be able to correctly identify the code clones in all cases. Consider two control flow graphs which are structurally identical but with entirely different node contents. Here checking the structural similarity alone is not sufficient to detect the presence of code clones. This is due to the presence of false positives in the detection of code clones. Therefore checking the structural similarity alone is not always enough to detect the code clones. Performing program slicing at the initial phase, constructing the control flow graph of the slices and checking the node contents of the CFG alleviates th defect present in the code clone detection up to a large extent. In our method we are using both static and forward dynamic slicing to generate program slices. Performing static slicing will display all the program statements which affect the value of a particular variable at a particular point. Here from the point specified, the rest of the source code is checked in a backward direction to get the result and the statements are checked in a bottom up manner. Using a text based or token based approach for clone detection has the defect of having to compare each and every line of the program code to detect the code clones [1]. By performing slicing as the initial step of code clone generation we can overcome this problem to a large extent, as slices display only the relevant parts of the program. This is one of the strong features of our approach.

C. Significance of the work

This section summarises the strengths of our approach. They are:-

- Uses Code Clones for Test Sequence identification
- Reduction in testing effort
- Need of Program Slicing in Code Clone detection
- Slicing avoids unnecessary program checking
- Applying slicing gives the relevant parts of the program
- Slices give dependence information in the program
- Static slices will give all possible Test sequences
- Interpreting of node values in Code Clones helps to generate correct test data values

VI. Conclusion

In our work, we have discussed the identification of test sequences from proper identification of code clones present in a program. We have also showed how program slicing is applied to detect code clones and the relevance of program slicing in code clone detection. Program slicing gives an overview of the relevant parts of a program and this reduces program tester's effort. Performing both static and forward dynamic slicing will give slice statements which gives a detailed view of variable dependency present in the source program. This in turn helps in test sequence identification. The control flow graphs of the slices are constructed and they are checked for structural and node content similarity in our method. This again strengths the code clone detection approach used here. Here we have provided a method to generate test sequence from code clones using program slicing. In future we are going to deal with more issues like

improvement in the detection of code clone methods and the type of slicing methods used in code clone detection.

References

[1] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical Report 541, Queen's University at Kingston, 2007

[2] J. Krinke. Is cloned code more stable than non-cloned code? In Workshop Source Code Analysis and Manipulation, pages 57-66. IEEE CS Press, 2008

[3] R. Komondoor and S. Horwitz. Using Slicing to Identify Duplication in Source Code. In SAS'01, pages 40–56, 2001.

[4] E. Juergens, F. Deissenboeck, B. Hummel, and S.Wagner.Do code clones matter? In ICSE '09, 2009.

[5]E. Juergens, F. Deissenboeck, and B. Hummel. Clone detection beyond copy & paste. In IWSC'09, 2009

[6] J. Krinke. A Study of Consistent and Inconsistent Changes to Code Clones. In WCRE'07, pages 170–178. IEEE CS, 2007

[7] M. Weiser, Program Slicing, IEEE Transactions on Software Engineering, vol. 10, no. 4, pp. 352-357, July 1984

[8] B. Korel, Automated Software Test Data Generation, IEEE Transactions on Software Engineering, vol.16, no. 8, pp.870-879, August 1990

[9] B. Korel, Computation of Dynamic Program Slices for Unstructured Programs, IEEE Transactions on Software Engineering, vol. 23, no.1, pp. 17-34, January 1997

[10] F. Tip, A Survey of Program Slicing Techniques, Journal of Programming Languages, vol 3, no3, pp. 121- 189, September 1995

[11] J. L. Chen; F. J. Wang; Y. L. Chen, An Object-oriented Dependency Graph for Program Slicing, Proc. Technology of Object-Oriented Languages, pp. 121- 30,September 1997

[12] K. B. Gallagher, J. R. Lyle, Using Program Slicing in Software Maintenance, IEEE Transactions on Software Engineering, vol 17, no. 8, pp. 751 – 761, August 1991

[13] B. Beizer, "Software Testing Techniques", Second Edition, International Thomson Computer Press, 1990, ISBN 1-85032-880

[14] C.K. Roy and J. R. Cordy. Near-miss Function Clones in Open Source Software: An Empirical Study. Journal of Software Maintenance and Evolution: Research and Practice, 23 pp., 2009