Assessing the Benefits of Incorporating Function Clone Detection in a Development Process

Bruno Laguë and Daniel Proulx
Quality Engineering Advisors
Information Technology Procurement
Bell Canada
2265 boul. Roland-Therrien
Longueuil, Quebec,
Canada J4N 1C5
Phone: (514) 448-5543
E-Mail: lague@qc.bell.ca
E-Mail: daproulx@qc.bell.ca

Ettore M. Merlo
Assistant Professor
DGEI Department of Electrical and Computer Engineering
École Polytechnique
P. O. Box 6079, Downtown Station,
Montreal, Quebec,
H3C 3A7, Canada,
Phone: (514) 340-5758
E-mail: merlo@rgl.polyml.ca

Jean Mayrand
Vice-President Technology
Telsoft Ventures Inc.
1000 De La Gauchetière Ouest
25th Floor
Montreal, Quebec
Canada H3B 3M4
Phone: (514) 397-8454
jmayrand@telventures.com

John Hudepohl
Manager, Software Reliability Engineering and Tool Development
Nortel Technologies
35 Davis Drive
Research Triangle Park,
NC 27709 USA
Phone: (919) 991-8412
hudepohl@nortel.ca

Abstract
The objective of the experiment presented in this paper is to bring insights in the evaluation of the potential benefits of introducing a function clone detection technology in an industrial software development process. To take advantage of function clone detection, two modifications to the software development process are presented. Our experiment consists in evaluating the impact that these proposed changes would have had on a specific software system if they had been applied over a 3 year period (involving 10 000 person-months), where 6 subsequent versions of the software under study were released. The system under study is a large telecommunication software. In total, 89 millions lines of code have been analyzed. A first result showed that, against our expectations, a significant number of clones is being removed from the system over time. However, this effort is insufficient to prevent the growth of the overall number of clones in the system. In this context the first process change would have add value. We have also found that the second process change would have provided programmers with a significant number of opportunities for correcting problems before customers experienced them. This result shows a potential for improving the software system quality and customer satisfaction.

Keywords: Software clones, software maintenance, software evolution, product assessment, source code metrics.

1. Introduction
In this paper, two or more functions are considered to be clones if they are identical or very similar. In a software system, it is highly possible that when a change is required in one of those functions, it may also be
required in the other copies. A high number of function clones in a software system could thus increase significantly the cost of maintenance. The reason is that if clones are not formally managed, problems already handled may be reworked several times. In a large system, this can be worsened by the fact that different people may handle these similar problems. In such a case, different solutions for fixing the problem may be implemented, each of them presenting a risk of introducing undesirable side-effects in the system. As a result, managing clones in a large system holds the promise of avoiding some maintenance costs and of reducing the risks of creating additional problems. It is thus an important aspect to be assessed for an organization when it is in the process of acquiring a large software product [6].

There have been many publications proposing various ways of identifying cloned components in a software system [1,2,9,10,11,12,13]. However, it is not clear how a clone detection technology can be applied in an industrial software development process in order to achieve significant cost savings. This paper addresses this specific objective.

The actual costs incurred from cloning vary depending on the amount of clones found in a system. The experience presented in this paper brings a valuable insight in evaluating the benefits that can be expected from the use of a clone detection technology during the development of a system. The experiment focuses on identifying and assessing the potential benefits of two process changes making use of a given clone detection technique, on a real large telecommunications system, over a 3 year period of time.

Section 2 describes the clone detection technology used in this experiment. Section 3 proposes two changes to the development process in order to take advantage of the clone detection technology. Section 4 gives an overview of the software system used for the study. Section 5 reports the results of the assessment of the two changes to the process, when applied on the system under study. Section 6 discusses special cases encountered during the experiment. Section 7 presents the conclusions that can be drawn from this experiment. Directions for future work can be found in section 8.

2. Selected clone detection technology

The clone detection technique we chose to use for this experiment is derived from a previously published one [13], as described here. Again, the goal of this paper is to assess the potential to achieve some benefits by introducing 2 process changes that attempts to exploit a clone detection technique, and not to assess that specific clone detection technique. Any of the published clone detection techniques could have been used for that experiment. We chose to use the clone detection technique described here mostly because it was readily available to us.

In [13], each function in a software system is characterized along its name together with 21 metrics extracted with Datrix™ [3,8] software metrics tool (see Appendix A for the list and description of these metrics). Then, any pair of functions can be compared on these characteristics, to yield a rating from 1 to 8 on an ordinal scale. When applied to the system under study, we have found the misclassification rate at Level 1 (also referred to as ExactCopy: a pair of functions is classified as an ExactCopy if all their metrics are equal, and both have the same name) to be null (all pairs were really exact copies). At Level 2 (also referred to as DistinctName: all the metrics of the two functions under comparison must be equal, but the name of the functions are different), the misclassification rate is below 1% as long as we do not retain functions that have no conditional statements. At higher levels in the ordinal scale (where some variations are tolerated in the metrics of the two functions), the misclassification rate grows higher. Even though many similar functions can also be found in these levels and should deserve attention in a more complete study, we chose not to include these higher levels in the experiment, in order to avoid overstated results.

Following the above classification of the relationships between functions, each function is itself classified as follows. If a function is involved at least once in a level 1 relationship, it is classified as an exact copy clone. If it is involved at least once in a level 2 relationship, but never in a level 1, it is classified as a distinct name clone. Otherwise, it is not considered as a clone in this study.

We define ExactCopy as the set of all functions classified as exact copy clones, and DistinctName as the set of all functions classified as distinct name clones. Finally, DistinctName’ is the set of all DistinctName clones which contains at least one conditional statement.

3. Modifications to the development process for tracking clones

In a previous paper [4], two changes to the development and maintenance process were proposed in order to take advantage of a clone detection technology. One was called Preventive Control, while the other was labeled Problem Mining. Both are summarized below for convenience.

3.1 Preventive Control

Preventive Control aims at keeping the introduction of clones under control, that is, to allow a new clone to
be added to the system only when good reasons exist for doing so. This first change to the development process can be described with the following 4 steps:

1. All changes submitted to the central source code repository are monitored.
2. When a change contains a new function, the function is compared against all existing functions in the system.
3. If the new function is a clone, the system architect is informed, and the programmer is queried to explain why a clone was created.
4. If the reason is not deemed acceptable by the system architect, necessary actions must be taken to reuse the original function.

This process was designed with the assumption that the underlying clone detection technology would detect identical functions as well as slightly different functions. For the system under study, we have observed that function clones classified in the ExactCopy and DistinctName sets are almost always totally equal to their matching counterparts. Hence, since we limit our experiment to these two sets, we assume that most of the newly introduced clones could have been avoided.

3.2 Problem Mining

Problem Mining is a process change that aims at coping with the existing base of software clones in a system already in service, for which new development and maintenance is still being done. This second process change consists of these steps:

1. All changes submitted to the central source code repository are monitored.
2. When a change contains a modified function, the previous (unmodified) instance of the function is compared against all existing functions in the system, and the list of all clones is built.
3. The above list of function clones is presented to the programmer.
4. The programmer determines whether the change must or must not be propagated in each of the clones.

With this approach, it can be expected that bug fixes and software enhancements are propagated in all contexts where the fix or the new functionality is required.

Again, since our analysis is limited to the clones found in the ExactCopy and DistinctName sets, we assume that propagation would be required in most cases.

Note that the two process modifications described here, using the clone detection technique of section 2, could be introduced at low cost in a context where software metrics are already captured on a regular basis, as the one described in [5].

4. The system under study

In order to assess the two process changes described in the previous section, we will study the impact they would have had on a software system if they had been applied over a 3 year period, where 6 subsequent versions of the software under study were released. It should be noted that this approach, which consists in studying past versions of the software system instead of performing a controlled trial, has no impact on the actual development process.

The system is a large telecommunication software. In total, 89 million of non-blank lines (including comments) were analyzed, for an average size of 14.83 million lines per version. The average number of functions per release is 187,000. The development effort over the 3 year observation period is on the order of 10,000 person-months. Note that the first version was derived from a large existing system. The language used is mainly procedural and similar to Pascal.

For each of the six versions of the system, each function has been classified along the cloning scheme described in section 2. The graph in Figure 1 shows that the overall ratio of functions classified in the ExactCopy and DistinctName sets stays at a relatively constant level over time, ranging between 6.4% and 7.5% of the overall number of functions in the system.

![Figure 1 - Ratio of clone types in the system.](image)

Since the system has grown from 170,000 functions in version 1 to 206,000 functions in version 6, the fact that the overall ratio of function clones remains stable implies that new clones are introduced in each release.

The above two factors indicate that the introduction of a clone tracking strategy in the development process could have a beneficial impact. This impact is analyzed in the next section.

5. Assessing the benefits of the two process modifications

In order to assess the two changes to the development process described in section 3, we will study the
evolution of the functions for each pair of subsequent versions.

In the following, we will frequently refer to the view shown in Figure 2.

![Figure 2 - Classification of functions in version N+1.](image)

When comparing the content of version N+1 to the content of version N in terms of functions, four sets are built:
1. New functions
2. Modified functions
3. Unmodified functions
4. Deleted functions

In this software system, a given function name can be used many times in different files within a version. However, file names must be unique. Hence, a function is uniquely identified by the couple (file_name, function_name). The following technique is used to build the above four sets of functions.

New functions are all couples (file_name, function_name) in version N+1 that do not exist in version N. Deleted functions are all couples (file_name, function_name) in version N that do not exist in version N+1.

All other couples (file_name, function_name) exist in both version N and N+1. We want to distinguish between modified and unmodified functions. The two subsequent versions of the function are compared on the basis of the same 21 DaTriX™ metrics used by the clone detection technology. When all the metrics are equal for the two versions of the function, the function is considered as unmodified. If any differences are found in the metrics, the function is considered as modified.

Please refer to section 6 for a discussion of special cases involved with the above technique that can introduce some function misclassifications, together with comments on their potential impact on the results.

Figure 3 shows the evolution of these 4 sets over the 6 versions of the software. All functions in version 1 are considered new.

![Figure 3 - Classification of functions in each version.](image)

It can be seen that functions are deleted in all versions, reaching an outstanding level (21,116) in version 6. Despite this fact, the overall number of functions in the system keeps increasing over time. Also, even though the level of activity is higher in terms of new and modified functions in versions 4 to 6, most of the functions in the system remain stable from one release to the next. This last point is of particular interest, since both of the process changes to be analyzed in the following subsections are triggered by additions or modifications of functions.

6. Benefits of “Preventive Control”

As mentioned in section 3, the Preventive Control scheme aims at avoiding the introduction of unnecessary function clones. We thus want to identify all ExactCopy and DistinctName’ clones that have been introduced in the system from version 1 to version 6. Furthermore, we want to analyze their evolution from the moment of their creation until version 6. Some will remain unmodified, others will be modified, and some others will be deleted. Note that when a clone is modified, the clone detection technology that we use in this experiment would not be capable of recognizing this function as a clone after the change. As a result, we expect to find here a larger number of clones in version 6 than what is possible to obtain from a single release analysis, as it was performed in section 4.

Four more sets of functions are created for tracking the evolution of the state of function clones between subsequent releases, as shown in Figure 4. The first set is composed of all new clones that the Preventive Control approach can identify. The three other sets keep track, in a cumulative fashion, of the evolution of clones identified in previous releases.

In specific terms, each set is built in the following ways.

First, each function known to be a clone in version N is reclassified in one of three sets in version N+1:
Deleted, Never Modified, or Modified once or more. Clones that were classified as deleted in version N remain in the same set in version N+1. A clone that was classified in the Never Modified set remains in this set if the function is still not modified in version N+1. This function may also have been deleted or modified. It would then be classified in the Modified once or more set, or in the Deleted set, respectively. A clone that was classified in the Modified once or more set in version N remains in this set in version N+1, unless it has been deleted, in which case it is classified in the Deleted set. A clone that was classified in the New set in version N is reclassified in the Modified once or more, Never Modified or Deleted set depending on whether the function is modified, unmodified or deleted in version N+1.

Finally, the set of new clones is built by comparing each of the new functions in version N+1 (Figure 2) with all other functions in version N+1. When the new function under analysis is involved in at least one cloning relationship (ExactCopy or DistinctName'), it is added to the New clones set. The other function in the relationship is also added in the New set if it is not already present in any of the other three clone sets of version N+1.

The results of this analysis are shown in Figure 5. In version 1, the set of New clones has been filled with all the functions that can be classified as ExactCopy or DistinctName' by performing a single release analysis. The other sets were left empty.

As shown in Figure 5, this cumulative analysis identifies 15,655 clones in version 6. This is to be compared to the 14,393 identified with the sole analysis of version 6, as it was done in section 4 to produce Figure 1. Even though the tracking of the clones over time did allow the identification of more clones, the increase is less important than expected.

### Figure 4 - Classification of clones in version N+1.

<table>
<thead>
<tr>
<th>Clones in Rel. N</th>
<th>Clones in Rel. N+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Clones</td>
<td></td>
</tr>
<tr>
<td>Modified once or more Clones</td>
<td></td>
</tr>
<tr>
<td>Never Modified Clones</td>
<td></td>
</tr>
<tr>
<td>Deleted Clones</td>
<td></td>
</tr>
</tbody>
</table>

### Figure 5 - Evolution of clones over time.

The cause of this unforeseen result lies in the deletion of many clones: the cumulative number of deleted clones from version 1 to 6 is 3,354. The authors are not aware of any specific rule applied by the programming community for removing clones, but it seems that some level of clean up is being done. However, this is still not sufficient to reach the point where the overall number of clones in the system would start to decrease. Also, many clones are still being added from version to version, such that the actions undertaken (if any) are actually oriented towards removing clones previously created than towards the prevention of their introduction. This observation also leads to conclude that the introduction of the Preventive Control scheme in the process would be beneficial.

It must be noted that most clones are never changed after their creation, as depicted by the relative size of the set of Never Modified clones in all versions. Also, only a very small number of clones have been changed at least once over the period of observation of this software system. However, we cannot conclude from Figure 5 whether or not that high stability in the clones is suitable or not. It is possible that changes made to the system should have been propagated to these clones even though it has not been done. The next section addresses this aspect.

### 6.1 Benefits of “Problem Mining”

At the moment where a software development organization would decide to adopt a clone tracking strategy in an existing line of products, the related source code would most probably contain some level of function clones. The Problem Mining scheme presented in section 3.2 aims at managing the set of existing clones. Whenever a modification is made to a function, programmers are given a list of functions where the modification may also be relevant. Each function hence presented to the programmers is an opportunity to fix a potential problem before its manifestation. In order to assess the potential benefits of the Problem Mining scheme, we want to compute two indicators for each release.
The first indicator is the number of opportunities presented to the programmers as described above. The second indicator is the number of changes already propagated in relevant function clones.

An upper bound and a lower bound will be computed for the first indicator. Consider a set of three function clones \{f_1, f_2, f_3\} in version N, and the case where \( f_1 \) and \( f_2 \) are both modified in version N+1. When \( f_1 \) is modified, \( f_2 \) and \( f_3 \) are presented to the programmer. When \( f_2 \) is modified, \( f_1 \) and \( f_3 \) are presented. The \( f_3 \) function would thus be presented twice to the programmer. It is possible that \( f_1 \) and \( f_2 \) were modified for different reasons. In such a case, it would be relevant for the programmer to consider the propagation of the modification each time \( f_3 \) is presented. However, it is also possible that \( f_1 \) and \( f_2 \) were changed for the same reason. In this case, it would not add value to present \( f_3 \) twice to the programmer (assuming that the right decision was taken on its first presentation). By computing the total number of functions presented to the programmers, we obtain the upper bound on the number of opportunities to propagate fixes. By computing the number of distinct functions shown to the programmers, we obtain the lower bound.

These numbers are computed as follows:

- Build the set of modified functions in version N+1 (Figure 3).
  - For each modified function \( f_i' \) in version N+1:
    - Take the characteristics of function \( f_i \) (the version of \( f_i' \) as it was in version N, that is, prior to its modification),
    - Build the set \( C_i \) of all clones of \( f_i \) in version N.
    - Compute the cardinality of \( C_i \).
    - Add the set \( C_i \) to the global set \( C \) of all functions presented to the programmers. Keep duplications.
  - Upper Bound is the total number of elements in set \( C \).
  - Lower Bound is the number of distinct elements in \( C \).

Tables 1 and 2 show the results of this analysis. Table 1 lists the lower and upper bounds on the number of opportunities to fix potential problems before they manifest themselves. It can be seen that the number of such opportunities is low in each release.

Table 1 - Bounds on the number of opportunities to fix potential problems.

<table>
<thead>
<tr>
<th>Version</th>
<th>Lower Bound</th>
<th>Upper Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>v2</td>
<td>74</td>
<td>183</td>
</tr>
<tr>
<td>v3</td>
<td>70</td>
<td>76</td>
</tr>
<tr>
<td>v4</td>
<td>445</td>
<td>655</td>
</tr>
<tr>
<td>v5</td>
<td>320</td>
<td>553</td>
</tr>
<tr>
<td>v6</td>
<td>407</td>
<td>875</td>
</tr>
</tbody>
</table>

Nevertheless, given that the costs of finding and fixing problems can sometimes be very high, and the low effort involved in verifying the lists of functions presented to the programmers, it can be expected that costs savings and increased customer satisfaction be achieved with Problem Mining.

Table 2 presents some statistics on \( C_i \), the number of clones presented to a programmer when a function \( f_i' \) in version N+1 is being modified. Only when \( C_i \) is greater than zero is a list of function clones extracted. The second column shows the proportion of time that a non empty list of clones is presented to the programmer whenever a function is modified. The two last columns give the average and maximum number of functions shown in these lists. It can be seen that most of the time, no list is built. When a list is built, it is generally small. Table 2 confirms that the additional effort required from designers to verify the lists of function clones is low.

Table 2 - Statistics of \( C_i \) (number of clones shown to programmer).

<table>
<thead>
<tr>
<th>Version</th>
<th>Modified functions having clones (( C_i &gt; 0 ))</th>
<th>Avg</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>v2</td>
<td>3.1%</td>
<td>2.51</td>
<td>6</td>
</tr>
<tr>
<td>v3</td>
<td>1.8%</td>
<td>1.49</td>
<td>7</td>
</tr>
<tr>
<td>v4</td>
<td>2.5%</td>
<td>1.65</td>
<td>11</td>
</tr>
<tr>
<td>v5</td>
<td>2.6%</td>
<td>2.13</td>
<td>8</td>
</tr>
<tr>
<td>v6</td>
<td>3.2%</td>
<td>2.32</td>
<td>17</td>
</tr>
</tbody>
</table>

The second indicator is the number of changes already propagated into relevant function clones. Consider a pair of clones \{\( f_1, f_2 \)\} in version N. If the same modification is made to these two clones, \( f_1' \) and \( f_2' \) in version N+1 will have the same characteristics in version N+1. The number of propagated changes is computed in a similar fashion as the previous indicator:

- Build the set of modified functions in version N+1 (Figure 3).
- For each modified function \( f_i' \) in version N+1:
  - take the characteristics of function \( f_i \) (prior to its modification),
  - Build the set \( C_i \) of all clones of \( f_i \) in version N,
  - Add the set \( C_i \) to the global set \( C \) of all functions presented to the programmers. Keep duplications.
- For each function \( f_j \) in this set:
  - if it is deleted in version N+1, add it to set D.
  - if it (say \( f_j' \)) is modified in N+1:
    - if \( f_j' \) is a clone of \( f_i' \),
      - add \( f_j' \) to Propagated set if it is not already there.

Table 3 thus shows the number of clones, among those that were presented to the programmers, that were deleted, and the number that were kept consistent with...
the function under analysis. All these cases represent opportunities already captured by the programmers, even if Problem Mining is not in place in this process.

Table 3 - Number of modifications in function clones already handled (clone deleted or change propagated).

<table>
<thead>
<tr>
<th>Version</th>
<th>deleted</th>
<th>propagated</th>
</tr>
</thead>
<tbody>
<tr>
<td>v2</td>
<td>0</td>
<td>48</td>
</tr>
<tr>
<td>v3</td>
<td>1</td>
<td>27</td>
</tr>
<tr>
<td>v4</td>
<td>51</td>
<td>214</td>
</tr>
<tr>
<td>v5</td>
<td>37</td>
<td>160</td>
</tr>
<tr>
<td>v6</td>
<td>16</td>
<td>234</td>
</tr>
</tbody>
</table>

When these numbers are compared with the lower bound on the number of opportunities shown in Table 1, it can be seen that about half of the opportunities are already seized. Nevertheless, since the other half remains, there are still potential benefits in Problem Mining.

The fact that certain changes are already propagated in some of the clones is interesting. We can speculate that programmers are aware of the clones they created themselves, or encountered in the course of their learning of the system. Cloning would thus be partly managed in this informal fashion. However, that approach seems to be insufficient to maintain all the clones in a consistent state, since many opportunities for propagating modifications still reside in the system.

7. Special cases encountered and their impact on this study

This section describes two special cases we encountered while performing the study, and discusses their impact on the results.

First, the sets described in Figure 3 are built by comparing the function names between subsequent releases. This name is actually the couple (file_name, function_name). In some cases, the name of a file or of a function may be changed. A function could also be moved from one file to another. In all cases, each affected couple would create a new function together with a deleted function. These misclassified new functions can also be mistakenly considered as new clones, but only if the original function was involved in a group of clones populated with 3 or more functions. Although these cases may have a significant impact in theory, we have observed that it is in fact negligible in this system.

Second, functions are assigned to the modified or unmodified set by comparing their metrics. Some minor modifications are thus not captured. We may thus consider some function clones to be “never modified” even if they were. Again, a large amount of observations make us conclude that this is not a problem in practice for this system.

8. Conclusion

We have presented the results of an experiment designed to assess the impact of two changes to the design process intended at taking advantage of a clone detection technology. The experiment has been performed by studying the potential impacts that the two modifications would have had if they were applied on 6 subsequent releases of a large software system.

Contrary to our expectations, we have observed a low rate of growth in the number of overall clones in the system, due to the fact that many clones are being removed from the system. This effort is nevertheless insufficient to keep the cumulative number of function clones in the system from growing. Preventive Control would help achieve this goal. Also, by allowing to avoid the introduction of clones to be later deleted, it would save the efforts of removing them later. We have also found that the second process change would have provided programmers with a significant number of opportunities for correcting problems before customers experienced them. This result shows a potential for improving the software system quality and customer satisfaction through an effective clone management strategy.

9. Future work

We have used in this study a very stringent clone detection technology. The framework for that type of study being in place, we intend to repeat the experiment with a clone detection approach that would also report slightly different functions, not only functions that can be classified in the ExactCopy and DistinctName sets. However, we want to keep the classification accuracy to a high level, in order to ensure that the programmers keep their confidence in the process. Hence, further work is required in order to select or adapt an existing clone detection technique with that objective.

We would expect the number of clones to be found to represent a significantly larger proportion of the overall system, and thus increase the benefits of the 2 process changes.

10. Acknowledgment

The authors would like to thank the École Polytechnique for its collaboration over the past years. Many thanks to our colleagues who are involved in product assessments, namely, François Guay and Claude Leblanc. Datrix™ is a trademark of Bell Canada.
11. References


Appendix A

This appendix lists the 21 metrics computed with the Datrix™ tool [8] that are used by the selected clone detection technique. These metrics are grouped by the three points of comparison of a clone relation.

Function metrics for layout:

- **ComDecVol**: Number of alphanumeric characters found in the comments located in the declaration section.
- **ComStrVol**: Number of alphanumeric characters found in the comments located in the executable section.
- **ComLogNbr**: Number of logical comments within a function. A logical comment is defined as a group of comments which are not separated by any token.
- **LocNbr**: Number of lines of code within a function. A line of code is defined as a line that is not empty. An empty line is a line which contains only white spaces, tabulations and a newline character.
- **VarLenAvg**: Mean number of characters of all variables that are used in the function. Unused declared variables do not qualify for the count.

Function metrics for expressions:

- **CalNbr**: Total number of call sites in the function. This metric takes into account repetitive calls to the same function.
- **CalUnq**: Number of distinct functions which are called by a given function.
- **CndCplAvg**: Arithmetic mean of the complexity of all the decisions in a function. The evaluation of the complexity of a decision is based on a token count. A value of 1 is attributed for constants and operators. A value of 2 is attributed for identifiers.
- **StmDecNbr**: Number of declarative statements within a function.
- **StmExeNbr**: Count of all the executable statements in the function. An executable statement is any portion of code where an assignment can take place.

Function metrics for control graph:

- **ArcNbr**: Number of arcs found in the control graph.
- **CndNbr**: Number of decisions in the control graph of a function.
- **CndSpnAvg**: Mean span of branches of conditional arcs. This metric is expressed in number of unit arcs.
- **KntNbr**: Number of arc crossings in the control graph. This includes the crossings that violates the principles of structured programming and those that do not.
- **LopNbr**: Number of backward arcs in the control graph of a function. Loops and backward goto cause backward arcs in the control flow graph.
- **NdsExtNbr**: Number of nodes in the control graph where the flow stops or returns to a calling software unit.
- **NdsNbr**: Number of nodes in the control graph.
- **NstLvlAvg**: Arithmetic mean of the nesting level of the control graph of a function.
- **PthIndNbr**: Number of paths in which the control flow of a function can follow between entry and exit nodes. It is a derivative of the Schneidewind metric [7]. Loops are visited only once in the same path.
- **StmCtlNbr**: Number of control statements.
- **StrBrcNbr**: Number of breaches of structure. It represents the number of arc crossings in the control graph of a function that violates the principles of structured programming.