Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics

Jean Mayrand
Vice-President Technology
Telsoft Ventures Inc.
1000 De La Gauchetière
Ouest
25th Floor
Montreal, Quebec
Canada H3B 3M4
Phone: (514) 397-8454
Fax: (514) 397-8451
jmayrand@telventures.com

Claude Leblanc
Information Technology
Procurement
Bell Canada
2265 boul. Roland-Therrien
Longueuil, Quebec,
Canada J4N 1C5
Phone: (514) 448-5091
Fax: (514) 647-3163
clleblan@qc.bell.ca

Ettore M. Merlo
Assistant Professor
Department of Electrical
and Computer Engineering
École Polytechnique
P. O. Box 6079,
Downtown Station,
Montreal, Quebec,
H3C 3A7, Canada,
Phone: (514) 340-5758
Fax: (514) 340-3240
merlo@rgl.polymtl.ca

Abstract

This paper presents a technique to automatically identify duplicate and near duplicate functions in a large software system. The identification technique is based on metrics extracted from the source code using the tool Datrix™. This clone identification technique uses 21 function metrics grouped into four points of comparison. Each point of comparison is used to compare functions and determine their cloning level. An ordinal scale of eight cloning levels is defined. The levels range from an exact copy to distinct functions. The metrics, the thresholds and the process used are fully described. The results of applying the clone detection technique to two telecommunication monitoring systems totaling one million lines of source code are provided as examples. The information provided by this study is useful in monitoring the maintainability of large software systems.

1. Introduction

Initial work for this study was conducted to identify freshman engineering students who were sharing too much of their software projects. When a student copies the work of another it is usually because he is not able to do the assignment or simply does not have time to complete it. Due to this lack of knowledge, capacity or time, the modifications made to the software are usually cosmetic. The foundation and structure of the software are rarely modified. Typically, variable names are modified, documentation is added or removed, source code layout is re-organized and function order in the file is changed. Knowing how near copies are produced, we have defined a process based on source code analysis and software metrics that identifies potential function clones.

The main goal of this work is to manage the growth in size and complexity of a software system due to source code cloning. The control of this growth is a concern for the telecommunication industry which places very high demands on software for reliability, longevity and modifiability.

Clone detection is a technique that finds functions that are an exact copy or a mutant of another function in the system. Previous work in clone detection using metrics was performed by [9], [10], [2] and [11]. Other approaches using text-based analysis were presented by [12] and [1]. The basis of comparison in this paper is a set of source code metrics measured on each function of a system. The scope of this work is procedural languages.

Most clones are created by copying a function and then making a series of modifications to the copy. We usually find this type of cloning in the absence of good re-use development processes. Designers generally copy an entire sub-system. Then they rename all the functions

1063-6773/96 $5.00 © 1996 IEEE

and start modifying the software. This technique ensures against unplanned effects on the original piece of code just copied. In the long run the software grows in size and complexity and requires more resources to maintain and enhance.

A large number of software clones induces undesirable side effects in a software system. The first possible effect is an increase in the resources required by the software on the system. This increases the cost of operation. For example, when software gets too big for a telecommunication system, new memory cards need to be acquired and deployed in the network. This represents an increase in cost. In the past, we have seen systems with up to 20% of their functions implemented as clones. This extra fat on the system exhausts hardware resource prematurely. The second effect of clones is to make the software more difficult to maintain. Problems solved in a function later re-appear in clones of an earlier version of the function.

The main difficulty in detecting clones in a large scale system is sheer size. Due to the system's size, it is impossible to manually track down the clones. Usually no documentation is kept on cloning activities. The main goal of our work is to identify these clones automatically.

Clone identification has great potential in the maintenance and re-engineering of legacy systems. The information obtained from clone detection can be used at the planning stage of major revamps of old systems. The information can also provide insight into latent difficulties in a re-engineering task. If there are many clones and these clones are a potential risk to maintenance, they could be removed prior to or as part of the re-engineering task.

This paper contains four major sections. Section 2 presents the source code assessment framework used to conduct this study. Section 3 describes the clone identification process; the four points of comparison are described, followed by the eight levels of cloning. Section 4 gives the results obtained using the clone detection process on two large scale systems. Finally, section 5 presents the cloning control process.

2. Datrix™ assessment framework

The assessment of the software was performed using Datrix™, a source code analyzer tool set [8]. Figure 1 presents the assessment framework of Datrix™. It is based on two successive abstractions of the source code. The first abstraction is from the source code to the Abstract Syntax Tree (AST). The AST is a tree-based representation of the tokens contained in the source code. It provides an exact representation of the source code. Datrix™ works with a number of source code languages. In order to support these languages, the AST is translated into an Intermediate Representation Language (IRL) [3]. The IRL contains four categories of information. The first category of information deals with the architecture of the software. This includes information on module, file and library dependencies. The second category of information about the software covers the static data types. The third category of information represents the control flow of the software. The last category represents the flow of data in the software. The IRL abstraction contains all the information required to compute metrics and to create graphical illustration of the architecture, data declaration, control flow and data flow graphs of the software.

This paper focuses on the control flow metrics and data flow metrics contained in the IRL. These metrics were selected because they provide detailed information on the internal characteristics of functions. Architecture and data declaration information were not selected since they provide no information about the internal characteristics of functions.

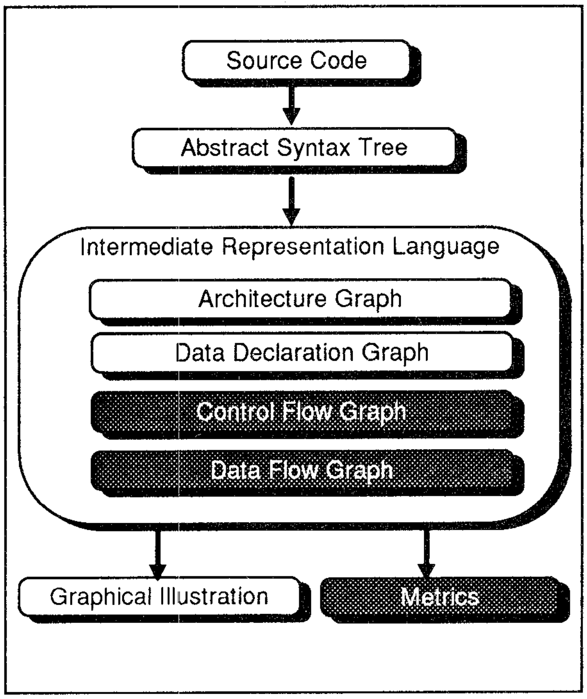

Figure 1 - Assessment framework

The IRL abstraction is described using the Object Modeling Technique (OMT) object model [4]. Figure 2 presents a simplified version of the IRL model used inside Datrix™ for abstraction of the control flow and data flow graphs. In this abstraction, both the expressions and the control flow between them are represented.

Expressions are the nodes *Nds* of the control graph *GphCtl*. The expression *Exp* has two specialized classes: the conditional expression *ExpCond* and the function call expression *ExpCall*. These classes are used to calculate metrics related to the number of decisions and the number of calls in a function. The associations *Use* and *Def* between *Exp* and *Ident* represent the data flow usage and definition of identifiers by the expression [13]. Relations between expressions are the *Arcs* of the control graph. Four types of arcs are defined. *CtlJump* represents an unconditional jump from one expression to another. The three other types of jumps are based on a Boolean or a switched decision.

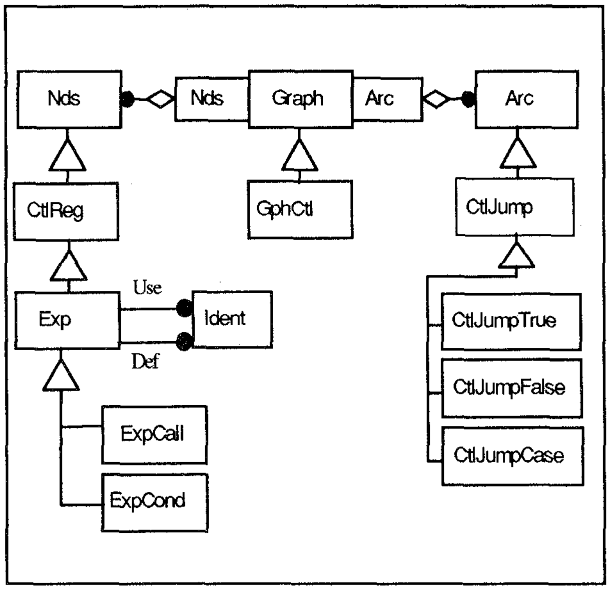

Figure 2 - Control flow and data flow model

Figure 3 presents an example of a C language function. This function contains one decision, two function calls, three variable definitions and two variable usages.

```
int fct( int param )
{
   int ret = 0;
   if( param != 0 )
   {
        fct2();
        ret = 1;
   }
   else
   {
        fct3();
        ret = 2;
   }
   return ret;
}
```

Figure 3 - C language source code example

Figure 4 presents the IRL translation of the function *fct* from Figure 3. The boxes represent nodes in the model. The token in the box states the type of node according to the model presented in Figure 2. The nodes START and END represent the beginning and the end of the function *fct*. The name after the node type is the identifier referred to by the node. Once this translation of the source code is obtained, it is possible to calculate metrics that are independent of source code language. The number of decisions in the function equals the number of IRL node of the type *ExpCond*. The number of functions that are called equals the number of IRL nodes of the type *ExpCall*.

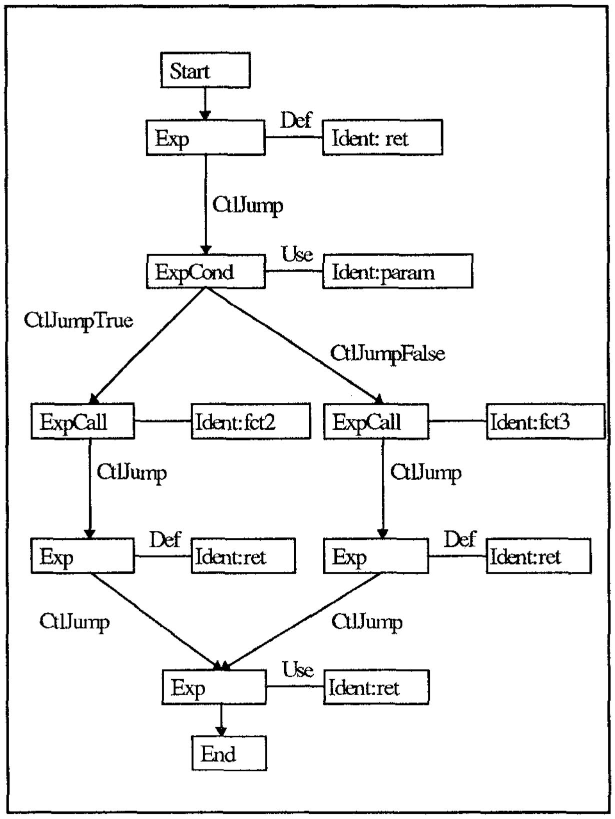

Figure 4 - IRL representation of the example

More than fifty metrics are calculated from the IRL representation. These metrics characterize files, classes and functions since this study aims at automatically identifying function clones, only the function metrics will be used. The metrics used in this study are described in Appendix A.

3. Clone identification

A clone pair is a pair of similar functions in a system. The number of potential clone pairs, considering pair

<f1,f2> as indistinguishable from <f2,f1>, for a system containing *n* functions is :

$$Potential\_Clone\_Pairs(n) = \frac{n(n-1)}{2}$$

3.1 Points of comparison for clones

This section describes the point-of-comparison concept and clone identification strategies. The identification of clones is based on the following four points of comparison:

1. Name.
2. Layout.
3. Expressions.
4. Control flow.

The first point of comparison between functions is their names. If two functions have the same name they are likely clones. In large scale systems, we have module boundaries that hide function names. These boundaries make it possible to have two functions with the same name in two different modules. The comparison of names is case sensitive. The names of functions in source code languages that are not case sensitive are translated into uppercase in IRL.

The second point of comparison is the layout of functions. We define "layout" as the visual organization of the source code, i.e. how the source code is organized in terms of comments, indentation, blank lines and variable names. Table 1 presents the metrics used to compare the layout of functions.

Table 1 - Layout metrics

Abbr.	Description	Delta
ComDecVol	Volume of declaration comments	10
ComStrVol	Volume of control comments	10
ComLogNbr	Number of logical comments	5
LocNbr	Number of non-blank lines	5
VarLenAvg	Average variable name length	2

For the layout, expression and control flow points of comparison, two functions can be considered equal, similar or distinct. Two functions are equal for a point of comparison if all metrics related to that point of comparison are equal in both functions. Two functions are similar for a point of comparison if the absolute difference is less than or equal to the delta threshold defined for each metric in the point of comparison. Two functions are distinct for a point of comparison if there is at least one metric where the absolute difference is greater than the delta value.

Deltas were defined on the basis of metric's definition and our knowledge of the distribution of that metric on large scale systems [6]. We have used numbers that are as low as possible in order to reduce the number of false accusations. A false accusation occurs when the clone detection process declares two unrelated functions to be clones.

Figure 5 presents an example of the evaluation of the layout point of comparison between four functions. Three symbols are used to represent the equal (=), similar (≈) and distinct (!=) relations.

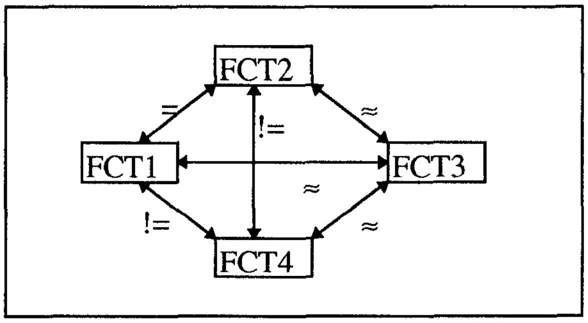

Figure 5 - Layout metric example

Table 2 presents the layout metric values of the four functions in the example. The FCT1 and FCT2 functions are equal since all their metrics are equal. FCT1 and FCT3 are similar since the absolute difference between their metrics are all below the delta values. FCT1 and FCT4 are distinct since the difference between the *VarLenAvg* metric values | 8.2-10.5 | = 2.3 is greater than the delta value of 2.

Table 2 - Layout metric value example

	Delta	FCT1	FCT2	FCT3	FCT4
ComDecVol	10	25	25	16	16
ComStrVol	10	50	50	55	55
ComLogNbr	5	6	6	9	9
LocNbr	5	12	12	16	18
VarLenAvg	2	8.2	8.2	9.3	10.5

The third point of comparison is based on the expressions in the functions. The number of expressions in a function, their nature and their complexity are considered. Table 3 presents the metrics for comparing expressions.

Table 3 - Expression metrics

Abbr.	Description	Delta
CalNbr	Total calls to other functions	5
CalUnq	Unique calls to other functions	2
CndCplAvg	Average complexity of decisions	2
StmDecNbr	Number of declaration statements	2
StmExeNbr	Number of executable statements	5

The fourth point of comparison between functions is their control flow. The control flow characteristics considered include number of nodes, number of arcs, information related to decisions and information related to loops in a function. Table 4 presents the metrics used in the comparison of function control flow. The delta tolerated is very low since a small variation in control structure has a large impact on the function's behavior. If the deltas are increased, the possibility of false accusations also increases. The delta value for the number of independent paths [7] is 100 since this metric increases rapidly when decisions are added sequentially in a function. For example, every time an **if** statement is placed at the beginning of a function the number of paths doubles.

Table 4 - Control flow metrics

Abbr.	Description	Delta
ArcNbr	Number of arcs	2
CndNbr	Number of decisions	2
CndSpnAvg	Average decision span	5
KntNbr	Number of knots	2
LopNbr	Number of loops	2
NdsExtNbr	Number of exits in a function	2
NdsNbr	Number of nodes	2
NstLvlAvg	Average nesting level	2
PthIndNbr	Number of independent paths	100
StmCtlNbr	Number of control statements	2
StrBrcNbr	Number of breaches of structure	2

These points of comparison are conceptually orthogonal and can be used independently. The following section will define an ordinal scale [5] of cloning based on a structured way of using these points of comparison.

3.2 Clone identification scale

We have defined eight strategies in identifying clones. These strategies define an ordinal scale of cloning. The first strategy is the most exacting one. It requires that the function be an exact copy without any modification. As we move up the scale, we identify function pairs that are less and less similar.

The ordinal scale is:

1. *ExactCopy*
2. *DistinctName*
3. *SimilarLayout*
4. *DistinctLayout*
5. *SimilarExpression*
6. *DistinctExpression*
7. *SimilarControlFlow*
8. *DistinctControlFlow*

The scale is ordinal since we have a monotonic increase in the difference between functions as we move up the scale. The scale is not an interval or ratio one since we have no the concept of distance between the values on the scale. For example, we cannot state that the distance between a *DistinctName* clone and an *ExactCopy* clone is the same as the distance between a *DistinctLayout* clone and a *SimilarLayout* clone. What can be said is that functions in an *ExactCopy* clone relation are more alike then functions in a *DistinctName* clone relation, the latter more alike then functions in a *SimilarLayout* clone relation, etc.

Figure 6 relates our concept of good and bad programming to the cloning scale. Best use is made of resources when all functions in a system are doing different things. The worst situation occurs when there are many copies duplicating functions in the system.

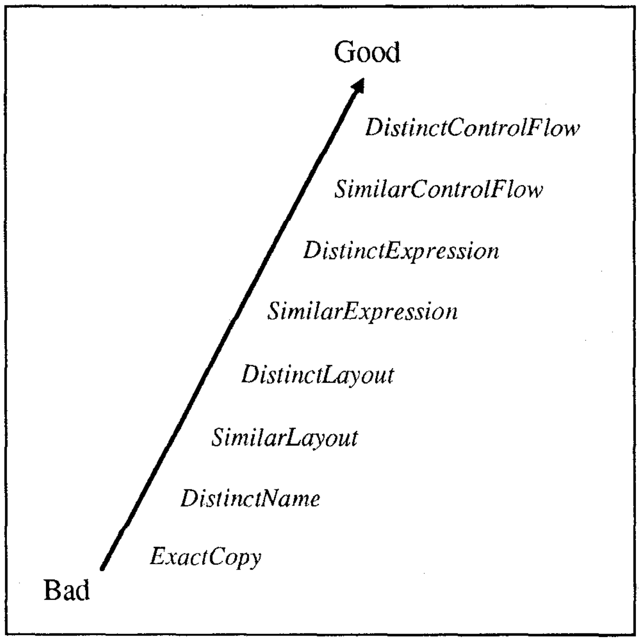

Figure 6 - Cloning scale

Table 5 presents the symbols used to represent the equal, similar and distinct relation for the points of comparison. Table 6 presents the mapping between the points of comparison and the cloning levels. The columns of the table are:

cloning scale,
point of comparison for name (Nam),
point of comparison for layout (Lay),
point of comparison for expressions (Exp),
point of comparison for control flow (Con).

Table 5 - Cloning scale symbols

Symbol	Description
=	Equal values for all metrics in a point of comparison between two functions
~	At least one metric not equal but within the delta in a point of comparison between two functions
!=	At least one metric not equal and outside the delta in a point of comparison between two functions
X	The point of comparison is not considered in the scale evaluation.

Table 6 - Cloning scale vs points of comparison

Scale	Nam	Lay	Exp	Con
1-ExactCopy	=	=	=	=
2-DistinctName	!=	=	=	=
3-SimilarLayout	X	~	=	=
4-DistinctLayout	X	!=	=	=
5-SimilarExpression	X	X	~	=
6-DistinctExpression	X	X	!=	=
7-SimilarControlFlow	X	X	X	~
8-DistinctControlFlow	X	X	X	!=

The first strategy is named *ExactCopy*. This strategy requires that all four points of comparison be equal. This means that all values of all metrics between the functions must be equal.

The second strategy is named *DistinctName*. This strategy is the same as the *ExactCopy* strategy except that the names of the functions must be different. This type of cloning appears when functions are copied inside the same module. The functions are renamed to avoid name clashes in the module.

The third strategy is named *SimilarLayout*. This strategy is the same as the *DistinctName* strategy except that variations are tolerated for the *ComDecVol*, *ComStrVol*, *ComLogNbr*, *LocNbr* and *VarLenAvg* metrics. This means that comments have been added or removed, the number of lines of code has changed or the variable names have changed. The main focus of this strategy is the layout of the functions. The constraint on the names of the functions is removed.

The fourth strategy is named *DistinctLayout*. This strategy requires that expression and control flow metrics be identical in both functions. This also implies that the layout is different.

The fifth strategy is named *SimilarExpression*. This strategy requires identical control flow metrics, but tolerates variation in the expression metrics. The constraints on layout are removed. This is the most typical form of cloning. It means that expressions are added or removed inside the structure of the function.

The sixth strategy is named *DistinctExpression*. This strategy requires that the control flow metrics be identical. Having the same structure but different expressions represents the re-use of a control flow pattern.

The seventh strategy is named *SimilarControlFlow*. This strategy tolerates variation in control flow structure. The constraints on expressions are removed. We must set a minimum difference in size and functionality in order to reduce the number of false accusations. This is because very small functions can look similar yet be very different. This is true, for instance, of functions that are strictly sequential without any control flow structure.

The eighth strategy is named *DistinctControlFlow*. This strategy captures all function pairs that are not considered clones or mutants according to our classification.

4. Case studies

This section presents the results obtained when the automatic clone detection strategies were applied to two telecommunication monitoring systems. These systems are currently being maintained and enhanced. The information provided by automatic clone detection is useful in improving the maintainability of the software. Clones can be removed and original functions placed in a re-usable library. The difference between two clones can be evaluated in order to parametrize a function and use this function in multiple contexts.

4.1 Analysis of clone relations

Table 7 presents the size, the number of functions and the number of potential clones for project A and project B. For project B, 227 functions were removed from the database. These functions were dummy functions introduced for configuration or database management. These dummy functions contained no executable code and were all alike. The database management functions handled user requests. The development team was aware of this and, in that specific context, using clones was the most efficient approach. These functions were removed from the metric database because they were artificially increasing the number of exact copy clones in the system.

Table 7 - Magnitude of case studies

Case Study	Lines of Code	Number of Functions	$\frac{n(n-1)}{2}$
A	506 823	7146	25 529 085
B	485 433	6645	22 074 690

The procedure used to identify clones in a system involves testing level 1 to level 8 for each and every pair of functions. Testing starts with level 1. If level 1 fails level 2 is tested and so forth up to level 8. The eight levels are mutually exclusive, i.e., a pair of functions can only be classified in one level.

Approximately 500 mathematical operations were required for testing each pair. The total cost of clone identification in projects A and B was 25 billion operations. The experiment was conducted using a C++ program on a Pentium 75MHz computer. The total time required to evaluate clones in each project was about 15 minutes.

Table 8 presents the number of pairs of functions in project A and project B corresponding to each strategy. Level 8, *DistinctControlFlow* captured all the pairs of functions that were not considered clones. This group represents the desired situation. In both projects, more than 96% of the relations between functions are not clone relations. This does not mean, however, that the system has almost no clones. It only means that if you take at random two functions, 96% of the time they will not be in a clone relationship. The clone relations are represented by levels 1 to 7.

Table 8 - Number of clone pairs per strategy

Strategy	Project A	Project B
1-ExactCopy	918	5 149
2-DistinctName	4 580	472
3-SimilarLayout	23 511	3 063
4-DistinctLayout	12 750	7 109
5-SimilarExpression	324 402	117 776
6-DistinctExpression	385 598	326 262
7-SimilarControlFlow	225 979	231 919
8-DistinctControlFlow	24 551 347	21 382 940

Figure 7 presents the relative cloning of project A and project B. The 100% on the Y-axis represents the total number of possible clone relations for a project. We used a relative scale to compensate for the small difference in size relatively between the projects. In general, the cloning is less frequent in project B. Only level 1 cloning is greater. We investigated this phenomenon and found that project B comprised multiple processes and the source code files contained a large number of functions. When a designer wants to create a new process, he copies the files of the original process but changes only a few functions. The bulk of the files remains intact, thus causing many level 1 clones.

The distinction between a level 1 clone and a level 2 clone is the name difference. A designer usually changes names when there is a clash at link time. These name clashes depend on the definition of the executable modules in the system. The differences between the two projects can be attributed to the architecture and the development technique used to add functionality to the system.

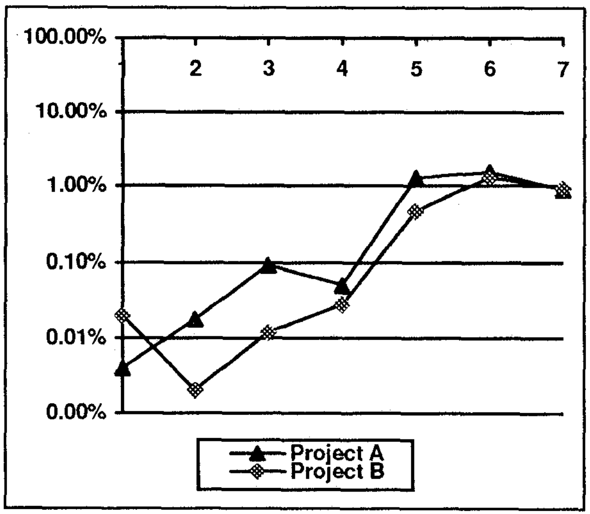

Figure 7 - Relative cloning

4.2 Function classification

The next question that comes to mind is: How many functions are implicated in a cloning relationship? To evaluate how many functions are in a cloning relation, we have to classify each function. Functions are classified according to their worst clone relation. The concept of worst is ranked from level 1 to 7, level 1 being the worst case. If a function has a level 1 cloning relation it is classified as a level 1 function. If a function has no level 1 cloning relation but has at least one level 2 relation, it is classified as a level 2 function. The functions classified as level 8 are those functions that have no cloning relation with any other function. Table 9 presents the results of function classification.

Table 9 - Function classification

Strategy	Project A	Project B
1-ExactCopy	1765	2849
2-DistinctName	588	219
3-SimilarLayout	500	606
4-DistinctLayout	394	306
5-SimilarExpression	1280	988
6-DistinctExpression	740	489
7-SimilarControlFlow	1158	840
8-DistinctControlFlow	721	348

Figure 8 presents the relative classification of the function cloning level. This chart indicates the types of cloning prevalent in the development of a system.

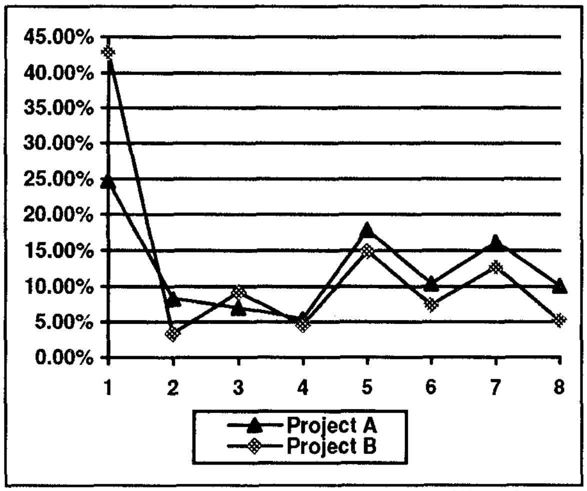

Figure 8 - Relative function classification

4.3 Visual validation of the case studies

A visual inspection of a sample of clones was conducted for each project. We started with level 1 clones forming the largest community. A community is defined as a group of functions in a cloning relationship. For instance, if we have three pairs of level 1 clones {(fct1,fct2), (fct2,fct3), (fct1,fct3)} we have a community of three functions (fct1,fct2,fct3). During the inspection of project A, we discovered that the level 1 functions were inside files copied into different directories. Table 10 presents the number and size of communities related to the file copy activities. Project A contained 1 function replicated in 5 different locations and 860 functions each replicated in 2 locations. In this case, the definition of reusable libraries is an easy task. By keeping a single copy of each function, 892 functions can be removed. This represents a 12% reduction in the number of functions in the system.

Table 10 - Project A level 1 communities

No. of communities	Size of community
1	5
4	4
8	3
860	2

During the level 2 inspection, we found one community of 7 functions all alike. The only difference was the name of the function itself or the name of the types and variables used inside it. This kind of cloning is more difficult to remove. One approach is to parametrize the function for types and variables. The difficulty in parametrization is directly related to the language's capacity. In languages like C and C++, parametrization is very easy; in other languages it represents a difficult task. The costs and benefits need to be assessed case by case. We also found that a size constraint, like the one imposed at level 7, would help eliminate very small functions. The metrics' discrimination power on small functions is greatly diminish. With small functions, we should change our strategy and use a text-based comparison.

For levels 3 to 7, the rate of false accusations increased. We had to rely on the inspection to judge whether cloning had occurred. The results should thus be used as a guideline, not a classification.

The procedure used in this visual validation can form the basis of an improvement program. The next section describes how to control the level of cloning in a software system.

5. Cloning control

The goal of the cloning control is to increase the maintainability of a system. The evaluation of the level of cloning provides a picture of the current state of the software. To modify this picture, we have to change how the software is developed and enhanced. Cloning control comprises four steps. These steps are describe in the next four sub-sections.

5.1 Measurement program

The first step is to implement a multi-version source code measurement program [6]. This provides source code metrics for the software system on a regular basis. This is a key factor in monitoring the modifications made to the software development procedures. Without this information it is impossible to evaluate the impact on the software product caused by changes made to the development procedures.

5.2 Design principles

The second step is to implement or review the design and programming guidelines in order to include policies regarding cloning. These policies must be part of the reuse strategies of the development organization. They need to cover the goals of reuse and acceptable practices in order to maintain a cohesive product architecture. Source code cloning could be tolerated under exceptional circumstances. For instance, during an emergency patch procedure when the entire integration test cannot be re-executed, cloning a module and patching the clone might be acceptable. In such a situation, a merge plan should be established for a later release. The goal is not to set up an inflexible process, but to proceed knowledgeably. Knowledge of the cloning activity and its rationale are very difficult to re-construct. This is why it is important to capture the information at the time the cloning is done.

5.3 Clone monitoring

The third step is to mandate someone in the development organization to monitor cloning in the system. This person is usually the system architect or the person responsible for integration. These persons are suitable since clone removal deeply involves the architecture and the libraries of the system. Monitoring on a regular basis should provide an indication of the addition and removal of redundant code. A minimal set of indicators are the percentage of redundant functions and the percentage of redundant statements.

Once the three steps are completed, the addition of new clones should be minimal. The final step involves the removal of existing clones.

5.4 Clone reduction

The fourth step targets clones that should be removed. This targeting should be based on the current enhancement of the system and the areas requiring a great deal of maintenance. Clone removal should start with level 1 clones. The cost of removing clones increases along the ordinal scale presented. Removing a level 1 clone is easier than removing a level 5 clone.

Level 1 clones are removed by creating common libraries of functions. This is a low cost and low risk technique. To remove clones from levels 2 and up, we use parametrization. The parametrization of a function can take many forms. It can be achieved by means of function parameters and/or preprocessing macros or, in the case of C++ source code, function templates. The selection of a specific technique is based on the nature of the cloning between functions.

Removing clones can become an expensive activity and hence, like any software project, should be well planned and well managed. For the first three steps, the cost is mainly toward the establishment of a systematic measurement program for the software. Once this program is in place, clone detection does not require extensive resources. The measurement program has further uses that should also be considered. For example, it is usually a key part of a product quality improvement program.

6. Conclusion

We have presented our experiments with automatic cloning detection. We have found this activity to be useful in improving the maintainability of a software system by managing and removing source code function clones.

The main cost in conducting the experiments was the measurement of the software. Cloning detection was not a major cost.

We have found the detection of level 1 clones to be reliable. Our visual inspection for the case studies showed a negligible level of false accusation. The level of false accusation increased substantially at level 3.

The metrics and deltas used influence classification. The addition of metrics for such domains as the interfaces and the reduction of delta values should minimize the number of false accusations.

Computational costs are polynomial. The projects presented were under 10,000 functions each. Our largest system under monitoring has 250,000 functions. We still have to optimize and enhance our strategy in order to apply it to this type of very large scale system. The first way of optimizing is to evaluate from level 8 to level 1 instead of level 1 to level 8. This would reduce the number of operations since most of cloning relation are level 8.

7. Future work

Our experiments leave many avenues to explore. The first would be to reproduce the experiment and work on the sensitivity of delta definitions. We found that both projects behaved very similarly. With the information at hand, it is not possible to deduce whether this is a coincidence or a lack of sensitivity of some clone detection strategies.

One avenue that we still need to explore is the use of relative delta instead of absolute delta. Instead of defining a delta of 2 for the number of decision, we could define a delta of 1% . Our hypothesis is that relative delta would help with small functions where metric values are very low. On large functions, the delta in percentage would produce large values. Probably a combination of percentage with a fixed upper value would be the best compromise.

The clone detection technique needs to be tested on object-oriented software. The detection of clones could be extended to the detection of patterns in object-oriented systems.

8. Acknowledgment

The authors would like to thank the École Polytechnique. Many thanks to our colleagues who are involved in product assessment, namely, Bruno Laguë, François Guay, Martin Leclerc and Alain April. Datrix is a trademark of Bell Canada

9. References

[1] Baker, S.B., "On Finding Duplication and Near-Duplication in Large Software Systems" In Proceedings of the Working Conference on Reverse Engineering 1995, Toronto, Canada, July 1995.

[2] Kontogiannins, K., DeMori, R., Bernstein, M., Galler, M., Merlo, E., "Pattern matching for Design Concept Localization", In Proceeding of the Second Working Conference on Reverse Engineering 1995, Toronto, Canada, July 1995.

[3] Mayrand, Jean , "Modélisation par niveaux sémantiques des programmes sources", (FRENCH), Master's thesis, École Polytechnique de Montréal, 1991.

[4] Rumbaugh, James; Blaha, Michael; Premerlani, William; Eddy, Frederick and Lorensen, William, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[5] Fenton, N.E. "Software Metrics -- A Rigorous Approach", Chapman & Hall, London, 337p., 1991.

[6] Mayrand J., Coallier F., "System Acquisition Based on Software Product Assessment", to be presented at the 18th International Conference on Software Engineering, Berlin 25-29 March 1996.

[7] Schneidewind, N. F., and Hoffman H., "An experiment in software error data collection and analysis", IEEE Transaction on Software Engineering SE-5, 3, May, 1979, pp.276-286.

[8] Bell Canada, "Datrix Reference Manual", 1996.

[9] McCabe T.J. "Reverse Engineering, reusability, redundancy: the connection", American Programmer 3, 10, October 1990, pp.8-13.

[10] Buss, E., et. al. "Investigating Reverse Engineering Technologies for the CAS Program Understanding Project", IBM Systems Journal, Vol. 33, No. 3, 1994, pp. 477-500.

[11] Kontogiannis, K., DeMori, R., Berstein, M., Galler, M. Merlo, E., "Pattern matching for Clone and Concept Detection", Journal of Automated Software Engineering, March 1996.

[12] Johnson, H., "Identifying Redundancy in Source Code Using Fingerprints", In Proceedings of CASCON '93, IBM Centre for Advanced Studies, October 24-28, Toronto, Vol.1, pp.171-183.

[13] Aho, Alfred V., Sethi, Ravi and Ullman, Jeffrey D., *Compilers: Principles, Techniques and Tools*, Addison Wesley Publishing Company, Reading, Massachusetts, 1986.

Appendix A

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.

LocNbr: Number of lines of code within a function. A line of code is defined as a line that is not empty.

VarLenAvg: Mean number of characters of all variables that are used in the function.

Function Metrics for Expressions

CalNbr: Total number of call sites in the function. This metriç 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.

StmDecNbr: Number of declarative statements within a function.

StmExeNbr: Number of executable statements within a function.

Function Metrics for Control flow 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 the branches of conditional arcs. This metric is expressed in number of unit arcs.

KntNbr: Number of arc crossings in the control graph.

LopNbr: Number of backward arcs in the control 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 the control flow.

StmCtlNbr: Number of control statements.

StrBrcNbr: Number of breaches of structure, based on the principles of structured programming.