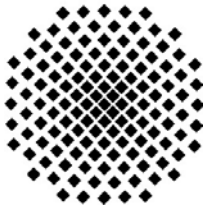

Universität Stuttgart

Fakultät Informatik, Elektrotechnik und Informationstechnik

Studiengang:	Informatik
Prüfer:	Prof. Dr. rer. Nat. Erhard Plödereder
Betreuer:	Dr. Rainer Koschke
begonnen am:	12.01.2004
beendet am:	11.07.2004
CR-Klassifikation:	D2.2

Diplomarbeit Nr. 2176

Semiautomatische Entfernung

des duplizierten Codes

Yidong Liu

Institut für Softwaretechnologie
Universität Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Table of Contents

Table of Contents 1
Table of Figures 5
Chapter 1 Introduction 7
1.1. Bauhaus 7
1.2. IML 8
1.3. Description of the task 11
1.4. Phases of the diploma thesis 13
1.5. Arrangement of the document 14
1.6 Expectance for the reader 15
Chapter 2 Code Clone Detection 17
2.1 Clone detection 17
2.1.1 Causes of code clone 17
2.1.2 Clone Types 19
2.1.3 Methods for clone detection 20
2.1.3.1 Method of Baker 21
2.1.3.2 Method of Mayrand et al 22
2.1.3.3 Method of Baxter 23
2.2 Clone Detection Tool – CCDIML 28
2.2.1 To be analysed system –concepts 29
2.2.2 Usage of CCDIML 29
2.3 Modification of CCDIML 33
2.3.1 Motivation of the modification 33
2.3.2 Specification of the modification 34
2.3.2.1 Storage of the clone pairs 35
2.3.2.2 New class for clone pairs 36
2.3.2.3 Storage by using attributes 37
Chapter 3 Code Transformation 41

3.1 Concepts of transformation 41
3.2 Examples of transformation system 44
3.2.1 TXL 44
3.2.2 The Munich Project CIP 46
3.3 Specification of code transformation 47
3.3.1 Background of transformation 47
3.3.2 Specification of Transformation 50
3.3.2.1 Specification of the Workflow 50
3.3.2.2 Specification of the transformation rules 51
3.3.3 Specification of the unparsing phase 56
3.3.3.1 Background of the unparser 57
3.3.3.2 Process of unparsing 60
3.3.3.3 Overload the visiting function 61
Chapter 4 Clone Replacement 65
4.1 Introduction of Emacs and Lisp 65
4.1.1 Emacs editor 65
4.1.2 Lisp 66
4.2 CPF mode in Emacs 67
4.3 Specification to enhance the CPF mode 69
4.3.1 Specification of the work flow for replacement 70
4.3.2 Specification of locations adjustment 73
4.3.2.1 Scene 1 of clone adjustment 74
4.3.2.2 Scene 2 of clone adjustment 77
Chapter 5 Implementation 81
5.1 Overview of implementation 81
5.2 Modification of CCDIML 82
5.3 Implementation of code transformation 85
5.3.1 Usage of CCR 85
5.3.2 Code transformation 86
5.3.3 Implementation of Unparser 89

5.4 Implementation of Clone Replacement 90
5.4.1 Implementation of the entry 91
5.4.2 Implementation of the new sub window 91
5.4.3 Implementation of the replacement 93
Chapter 6 Summary 95
6.1 Problems in Bauhaus 95
6.1.1 Package IML_Attributes 95
6.1.2 Result of the clone pair 96
6.2 Conclusion 97
Glossary and Abbreviations 99
Bibliography 100

Table of Figures

Figure 1: Old Version of IML Hierarchy 9

Figure 2: New Version of IML Hierarchy 10

Figure 3: Samples of the clone types 20

Figure 4: Comparison of the ASTs 24

Figure 5: Basic Subtree Clone Detection Algorithm 25

Figure 6: Example of clone sequence 26

Figure 7: Sequence Detection Algorithm 27

Figure 8: Detecting more complex clones 28

Figure 9: Usage of CCDIML 30

Figure 10: Flowchart of CCDIML 32

Figure 11: Format of the CPF file 33

Figure 12: Procedure of using IML-Attribute 37

Figure 13: Modification of CCDIML 38

Figure 14: Transformation Rules 41

Figure 15: Transformation System 42

Figure 16: Precondition of transformation system 43

Figure 17: Implementation of transformation rules 43

Figure 18: Transformation for Code Clone Replacement 49

Figure 19: Work flow of transformation 52

Figure 20: Example for unparsing 58

Figure 21: Process of unparsing 59

Figure 22: Sample of overloading functions from Visitors 62

Figure 23: CPF navigator 68

Figure 24: Clone view frame with Type I clone 69

Figure 25: Flowchart of clone replacement 71

Figure 26: Clone Adjustment Scene 1-1 74

Figure 27: Clone Adjustment Scene 1-2 75

Figure 28: Clone Adjustment Scene 1-3 ..... **75**

Figure 29: Clone Adjustment Scene 1-4 ..... **76**

Figure 30: Clone Adjustment Scene 1-5 ..... **76**

Figure 31: Clone Adjustment Scene 2-1 ..... **77**

Figure 32: Clone Adjustment Scene 2-2 ..... **78**

Figure 33: Clone Adjustment Scene 2-3 ..... **79**

Figure 34: Clone Adjustment Scene 2-4 ..... **79**

Figure 35: Clone Adjustment Scene 2-5 ..... **80**

Figure 36: Structure of implementation **81**

Figure 37: New class for clone pair **83**

Figure 38: Flowchart of saving the clone list as attribute **84**

Figure 39: Usage of CCR **85**

Figure 40: Unparsing class Return_With_Value **90**

Figure 41: Select definition file of function/macro **91**

Figure 42: New clone view frame **93**

Chapter 1 Introduction

This document is concerned with the diploma thesis (Diplomarbeit) "Semi automatic elimination of code clones". In order to get a preparatory impression, the Bauhaus project should fist be presented briefly as background in this chapter; following is the description of the task of my work; finally an overview of the structure of the work will be specified.

1.1. Bauhaus

Bauhaus is a project by the “Institute for Software Technology, Compiler Group, University of Stuttgart Germany”, in collaboration with “Fraunhofer Institute for experimental software engineering (FhG IESE)”, Kaiserslautern.

The Bauhaus project supports the maintenance of software; in particular within the range of the program understanding, it wins the DoIT Award 2003 for outstanding scientific achievements and problem solutions in the field of working and software in the enterprise and institution of the research establishment or universities in German State Baden-Wuerttemberg.

It is well known that 60-80% of the costs of a software product arise after its first delivery according to various investigations, i.e. the amount of costs is used for current maintenance of the software for the error correction and adjustment to changed requirements.

Studies show that engineers spend far more than the half of their work time to understand the program first, which is to be changed, before they go about to design

and realize concrete changes. As a consequence of lacking information about the architecture of the existing system, a lot of modifications would always be performed, which ignore the original designing principles of the system, whereby the software becomes nontransparent, bulked and has difficulties in maintenance.

No matter what the reason of maintenance is, it is the task of the engineer, to carry out the maintenance of the software. This can however always stand before a nontrivial task, because:

- The maintenance is not necessarily executed by the programmer of designer of the software
- The documents of the software could either not exist at all or already be obsolete
- The architecture of the software could have been washy by already accomplished work of maintenance

The maintenance engineer lacks the information of the actual status of the software system in either case, which is to be maintained. In practice it is very improbable that the engineer is perfectly informed about the structure of the software. So he needs to procure the information somehow by himself, which are often the principal part of the maintenance costs.

1.2. IML

As introduced in the above section, the first goal of Bauhaus project is to develop the mean of description, which describes the software representation of the whole system. The means of description in Bauhaus is called IML (**I**nter**M**ediate **L**anguage).

The complete graph, which is to be created to present the intermediate language, will be called IML_Graph in the following. It consists of sub graphs HPG (Hierarchical

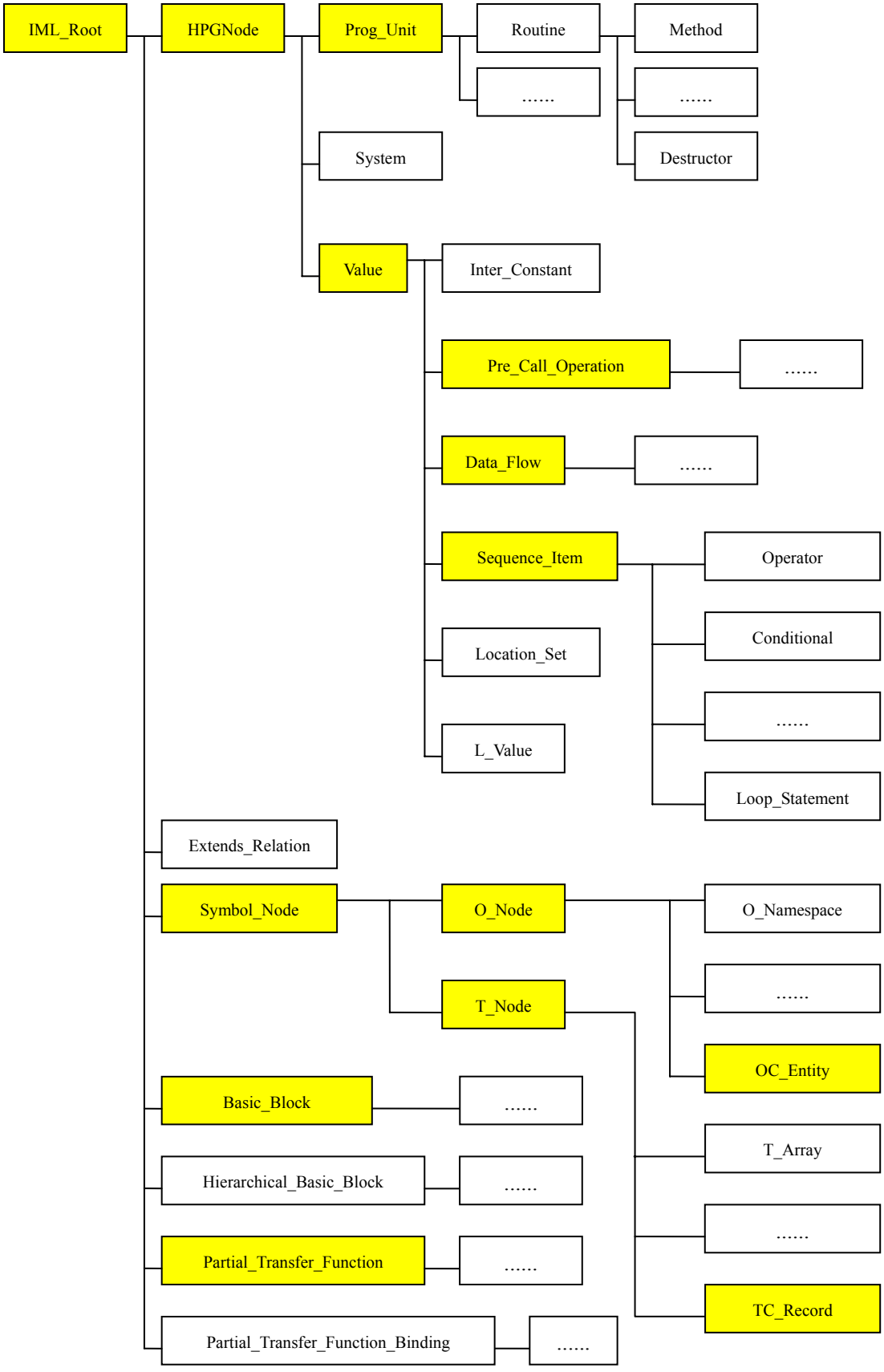

Figure 1: Old Version of IML Hierarchy

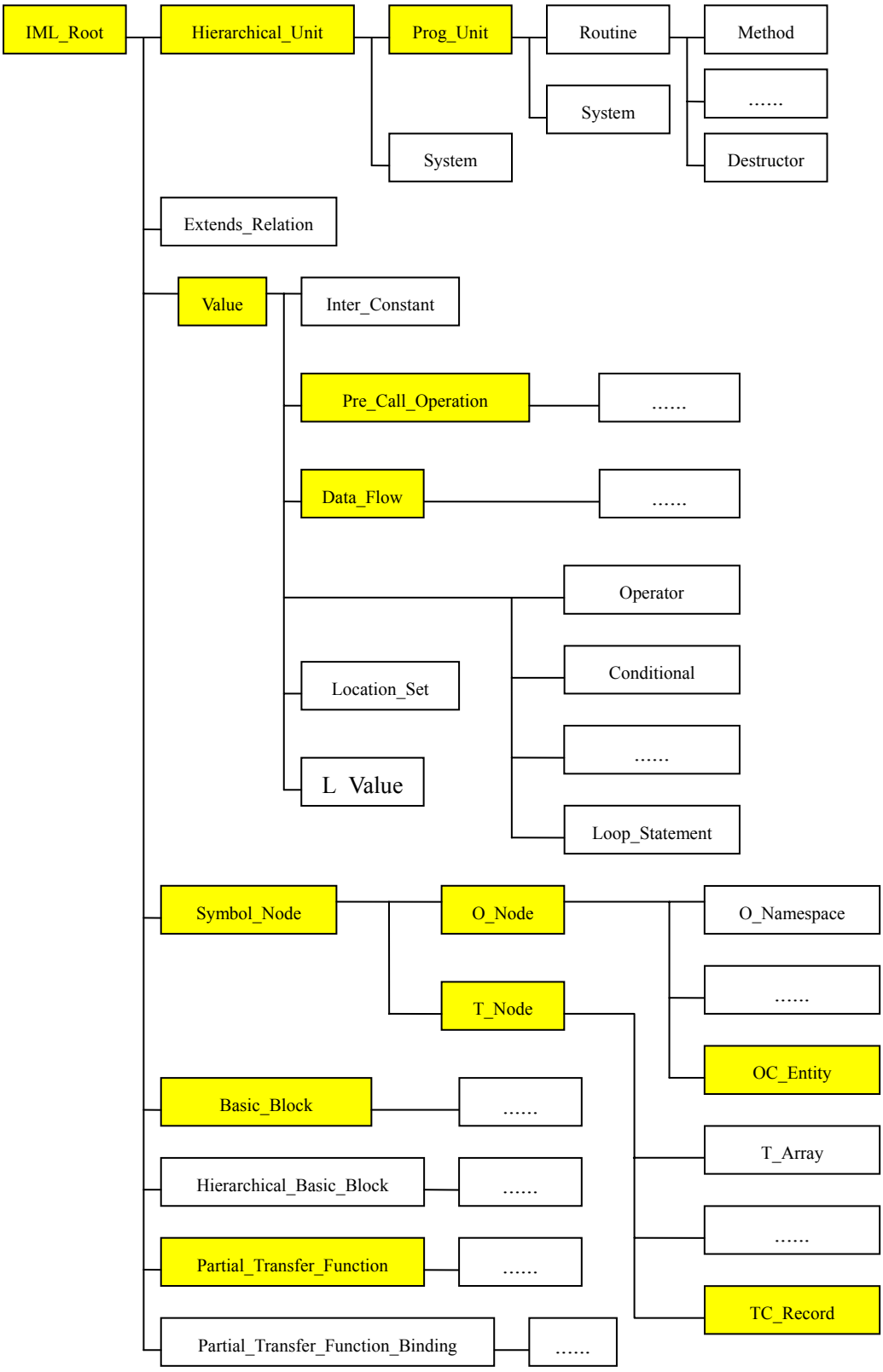

Figure 2: New Version of IML Hierarchy

Program Graph) and additional nodes for objects and types, which are used for the presentation of its own symbol table.

The IML-Graph consists of nodes and edges. The nodes represent the objects with definitive class in real time, and each node has a few attributes and methods of its own class. The attribute is either reference of other node or contains special value, and the method can execute some given function for the accordant class. The edge of the IML-graph is a special type of attribute as a result, namely pointer.

The IML nodes type has a inheritance hierarchy; the top of the hierarchy is the class IML-Root, and all the other class are derived from it. The specification of the IML has been partly changed during my work. (see figure 1 and figure 2)

In the new version of specification, the class Value will not inherit from the *HPGNode*, but directly from the *IML_Root*, and the class *Sequence_Items* is from the IML-Hierarchy deleted, all classes under class of *Sequence_Items* are derived from class Value inherited. While the whole specification is well structured, the change of the IML hierarchy has almost no influence on my source code, which was developed before the new version of specification. In my diploma thesis, I modified the source code in the code clone detection tool CCDIML: in the original code of CCDIML, the class *Sequence_Items* is used for a lot of sequence class, but this class-type is deleted in the new specification, all classes inherited from *Sequence_Items* are now directly inherited from class *Values*. Therefore a few manual modifications should be accomplished in the source code of CCDIML. To be modified are only the class name, the data structure and algorithms remains unchanged.

1.3. Description of the task

The Bauhaus toolbox contains a tool, with which the duplicated code can be found automatically. This tool is a command line tool– CCDIML (**C**ode **C**lone **D**etection on

IML), it uses the method by Baxter el (ICSM 1998) and compares abstract syntax-trees. The abstract syntax-trees in the Bauhaus are represented in form of the IML, an internal intermediate language for program analyses in Bauhaus.

In this diploma thesis, the possibility of a semi automatic replacement for duplicated code (Type-1 and Type-2 of clones) should be created. As an alternative for the user, the duplicated code should be replaced with macro- or function calls. The realization for the definition of the macros and functions must be based on an analysis in the IML found duplicated abstract syntax-trees, which contains the necessary information of the represented code. The output of this analysis is the code of definitions for the macro- or function with suitable formal parameters as well as that of the macro- or function call, which would be used to replace the duplicated code with appropriate actual parameters. This procedure of replacement of the duplicated code is to be called code transformation.

The above-mentioned transformations should be processed only in text, i.e. the transformation is made directly in program text and not on the abstract syntax-trees. This restriction is justified for that the program text from the IML cannot be recovered original-faithfully, in particular if macros and comments are used.

Over the above, a tool should be developed, which allows the user to use the above described transformations interactively. The user should have the possibility of examining, adopting, modifying and rejecting individual transformations. In particular the possibility must be given, that reasonable name for the new created macros and functions can be assigned. The tool must record all decisions of the user and allows user to cancel individual transformations.

The tool is to be developed in such a way that it can be used later also for other code transformations. The recognition of the duplicated code is not part of the task, instead the Bauhaus clone detection is to be used.

A high quality of the documentation, the design and the implementation as well as to an extensive test are expected. All program documentations are to be written in English.

Implementation tools:

Ada95 - compiler Gnat and Bauhaus developing environment.

Emacs-lisp – the language used to extend Emacs.

1.4. Phases of the diploma thesis

In order to get an overview of the work, which is to be accomplished within my diploma thesis, a brief description of some important phases are delivered here.

Practice into the described task: This phase includes the literature research and practice of the described task. As an introduction, the two papers ([IM98] [IML03]) about the specification of IML should be in part analysed.

Build up the plan for the task: After basic practice in the theme, the project plan should be built up, which is used to control the progress of the task.

Build the specification of the code transformation & replacement: There are three steps in this phase: first is to modify the existing tool – CCDIML, the result of the clones, which were found by CCDIML, should be saved into the IML-file and may be retrieved from it again; the second part is the main task, code transformation, which is to create new functions/macros for the found clone-pairs; the last is to modify the existing CPF display mode, which can only show the result of the found clones, to enable the clone replacement.

Coding and test: In this phase, the above specification should be realized and implemented, and finally be tested.

Documentation of the work: During the period of working, all the procedures should be recorded and at last accomplished as an integrated composition.

1.5. Arrangement of the document

This paper is arranged in the following chapters:

- **Chapter 1**

  It introduces the work; including Bauhaus Project, IML as an intermediate language for the description of software programs in Bauhaus, description of the thesis, phase of working for the thesis, arrangement of the documents and exception for the reader.

- **Chapter 2**

  The code clone detection is described in chapter 2. First is the foundation of clone detection, then the tool CCDIML briefly introduced. At last is the specification of the modification of CCDIML, the aim of the modification is to save the clone list in *IML_Graph* for further analysis.

- **Chapter 3**

  This chapter is about the code transformation. It introduces the basic theory and some practical tool/project of transformation. This is the main segment of my work. The specification of the clone code transformation is described in the last section. As part of the transformation, the unparser is also introduced and enhanced to meet the demands for recovering the source code.

- **Chapter 4**

  This chapter introduces how to display the new functions/macros and replace the clone code with a call of the function/macro. It works on the existing CPF mode, which shows the clone code in Emacs. The specification of displaying and replacement is described in the last section.

- **Chapter 5**

  As in the above chapter specified, the implementation of the three parts is introduced in this chapter: save the clone list in IML_Graph, transfer them into functions/macros, then display and replace the clone code

- **Chapter 6**

  In this chapter, some problems are listed, which were appeared during my work.

These problems should be corrected to improve the result of code transformation. Finally the work is summarized

1.6 Expectance for the reader

The author expects that the readers of this document have the following knowledge:

- Knowledge of syntax and semantics of the program language Ada95 (see [Ada95])
- Knowledge of syntax and semantics of the program language C (see [KR88])
- Knowledge of syntax and semantics of Emacs Lisp (see [Elisp])
- Global knowledge of the object-oriented methods and the object-oriented programming language

Chapter 2 Code Clone Detection

The code clone detection is to be introduced in this chapter, for that it is the base of the code clone replacement. The source code will be analyzed with the tool of code clone detection, and then the result of the clones is to be saved, which will be analyzed later to create new functions/macros for the clone code.

In the following sections, the theory of clone detection is to be introduced first, and then the code clone detection tool – CCDIML (see [CCDIML]), at last is the modification of the CCDIML, in order to enable the analysis of the found clone pairs for the code transformation.

2.1 Clone detection

Data from previous research [Lague97 Baker95] suggests that a considerable fraction (about 5-10%) of the source code is duplicated code (code clone). This kind of software clones appear for a variety of reasons, mostly the programmers might reuse the existent code to meet their current demands, just copy the similar source code and paste them into somewhere and do some modification to customize the copied code to the new context.

2.1.1 Causes of code clone

Legacy code is constructed by less structured means. Especially, the existing code could have been ad hoc reused for a considerable amount of code, most of the editors for programming have the basic-function or hot key like “copy” and “paste”, which hasten the ubiquity of the clone event. The "Copy & Paste" act makes the clone codes

total duplicates of the same function, it seems to simplify the work of the programmers, they could just pick up the existent implementation of some abstraction and reuse them, but it causes some following maintenance problem that is the principle of encapsulation of software engineering could be broken. Many systems have been found with poor copies of insertion sort on different arrays somewhere around the code. Such clones are an indication that the data type operation should have been supported by reusing a library with public function or macro rather than pasting a copy.

In another case, an abstract module offers a general solution for some kind of computation, this module will be reused anywhere it is needed, and in order to meet some individual request, it should also be partly modified. The act of copy and modify might cause a near-miss clone; the code is almost the same as the original except that they have few irrelevant order and variable name. They both have the same semantic and should also be treated as code clone.

Furthermore, some clones exist for legitimate reasons. Systems with tight time constraints are often hand optimized by duplicating frequent computations, particularly when a compiler does not offer in-lining of arbitrary expressions or computations.

At last, there exist some occasional codes, which are not per “Copy & Paste” generated, but just accidentally identical with each other. In fact they are not code clone, result of the full investigation shows that such obvious clones are not intended to carry out the same computation. When the size of the system goes up, the number of accidents of this type drops off dramatically. Therefore, this kind of occasional code should be ignored in clone detection.

Disregard the last case of accidental code, all above mentioned clones will increase the mass of code. It brings on a great deal of unnecessary maintenance and inspection

for the programmers, which increase the working time and cost of software developing certainly. As a result for this problem, such clones (real clones) could be converted to a prototype (in function or macro) first, and then replaced by calling the function/macro, which furthest reuse the implementation of abstractions and reduce the maintenance of system without breaking the software engineering principle of encapsulation.

The code clones in the program lead to some problem and inconvenience as following:

- High cost for maintenance, program understanding and test
- Difficulty for error correction
- Difficulty for changing source code
- Redundant data to be analyzed and visualized

2.1.2 Clone Types

As in Bauhaus project defined, there are three types of code clones according to the similarity of the source code, namely Type I, Type II and Type III.

Type I clone is to describe the code segments that are copied exactly from one to another without any change in the implementation. It is the typical clone of the act “Copy & Paste”.

Type II clone is to describe the code segments that are copied from one to another with renaming of some parameter. The implementation and structure of the both segments remain the same, but only some identifiers are renamed. E.g. reuse an existing function, and manually change the name of function or some parameters.

Type III clone is to describe the code segments that are first copied from one to another, and then the copied segment will be further modified, not only the identifiers in the code, but also the implementation or date structure. E.g. the base function

should be extended after copying.

There is also a special type of clone, namely Type IV. It describes the code segments, which are not copied from each other. The code segments have different implementation and date structure, but they just achieve the same function. E.g.

- Code segment 1: *iSum = iSum + 1*;
- Code segment 2: *iSum++;*

This type of clone is as a concept defined in present, and it will not be processed in the tool of clone detection. (CCDIML)

The following are the samples of each type of clone:

i = length(I) if (i<3){…}	i = length(I) if (i<3){…}	Type I
i = length(I) if (i<3){…}	k = length(j) if (k<x){…}	Type II
i = length(I) if (i<3){…}	k = length(j) if (j<3){…}	Type-II with inconsistent renaming
i = length(I) if (i<3){…}	k = length(j) if (k>x){…}	Type III
X = X + 1	X++	Type IV

Figure 3: Samples of the clone types

2.1.3 Methods for clone detection

In order to find out the duplicated code (Code Clones) in the program, some methods are developed to solve the problem. The most well-known methods for clone

detection are from Baker [Baker95], Baxter [Baxter98] and Mayrand et al.[REEG03] Among the above-mentioned methods, the method of Baxter is adopted as the basic theory for the implementation of the code clone detection tool CCDIML in the Bauhaus project. In the following section, the methods of Baker and Mayrand et al. will be briefly introduced and the method of Baxter will be described in detail.

2.1.3.1 Method of Baker

The Baker's method is based upon lexemes. It generates a symbol chain for each line in the source code from parameter symbols and non-parameter symbols (called Parameter-String or P-String), the symbol chain has the following properties:

- Structure of the line should be mapped to an unambiguous non parameter symbol (functor)
- Identifiers should be collected in the argument list
- The result is the argument list of functor
- Sample: x = x + y → (P = P + P; x,x,y) →α x x y

The concatenation of the P-Strings of all the lines in the source code represents the whole program, the method of Baker traverses the source code by comparing the P-Strings in the P-Suffix-Trees to find out the P-Match, which represents the code clones in the program.

The method of Baker is based upon lexemes and runs very fast, the expression of program will not be changed by insertion of blank lines and comments, and it is independent of the programming language. Thus, it will also miss out on something:

- the equivalent expression with commutative operations

x = x + y	x = y + x

- the same sequences of assignment, but in different lines

if (a>1) {x=1}	if (a>1) {x=1}

- the same sub expression

if (sp>0)	if ((sp>0)&&(s[sp]!=a))

2.1.3.2 Method of Mayrand et al.

The Mayrand et al. is based upon metrics of the source code. The identifiers of the source code are ignored from the metrics. The expected comparison of code segments looks like: (code1 = code2) ←→key-data (code1) = key-data (code2).

The aspect of the key data consists of four parts: ***name, layout, assignment and control flow***.

The ***name*** indicates:

- The relative number of the joint symbol;

The ***layout*** indicates:

- The number of symbols of comments (including declaration and implementation)
- The number of multiline comments, the number of non-blank line (includes comment),
- The average length of the symbol;

The ***Assignment*** indicates:

- The entire number of the calls of function
- The number of different calls, the average complexity of the decision in the function
- The number of declaration
- The number of executable assignment

The ***control flow*** indicates:

- The number of edges in the control flow graph (CFG)

- The number of the nodes in the CFG
- The number of the conditions in the CFG
- The number of the path in the CFG and the complexity metrics about the CFG.

To compare two functions f1 and f2 in respect of one aspect:

- Same: they have the same value of metrics
- Similar: all the values of the metrics lie in a certain range
- Different: at least on metrics value lies outside the range

The classification of potential clone pairs is according to the result of the comparison of the aspects. Two sections of source code are Type I clone, if they are the same in each aspect; the sections are Type II clone, if they have the same aspects in assignment and control flow, and the aspects in layout and name are similar; in another case, if the two sections have different aspects in name and layout, but the same aspects in assignment and control flow, they are also Type II clone; the last case, if all the aspects are different, the two sections are of course different, they are not code clones.

The method of Mayrand et al. is based upon the metrics of source code, the aspects are not independent with each other, and the definition of the range is necessary. Moreover, the classification is not complete and the precision the result depends upon the set of selected metrics.

2.1.3.3 Method of Baxter

Different from Baker and Mayrand et al., the method of Baxter is based upon the AST (Abstract Syntax Tree). As a first step in the clone detection process, the source code is parsed and an AST is produced. After that, three main algorithms are applied to find code clones. The first algorithm is the basic one, finding sub tree clones, whose purpose is just to detect sub tree clones. The found clones are single clone pairs. The

second one, sequence detection algorithm, is concerned with the detection of variable-size sequences of sub tree clones, and is used essentially to detect statement and declaration sequence clones. The last one, clone generalization algorithm, which is to remove the sub clones in the sequence and combine the sequence of clones into a complete clone.

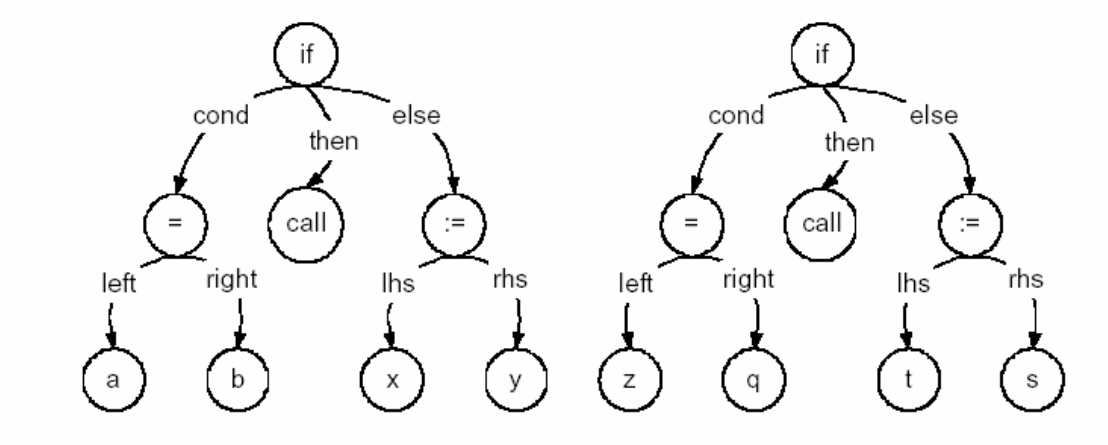

Figure 4: Comparison of the ASTs

Finding Subtree Clones:

To find the subtree clones is in principle easy; just compare every subtree to every other subtree, in order to find the equality. Actually, several problems, like near-miss clone detection, sub-clones and scale, will arise. We handle near-miss clone by comparing subtrees for similarity rather than exact equality. The sub-clones problem is that the large clones with a sequence of sub-clones should be recognized and be handled, i.e. all the detected sub-clones in the sequence need to be eliminated and a large clone will be generated. The scale problem is harder. For an AST of N nodes, this comparison process is O (N^3), and, a large software system of M lines of source code hast N = 10*M AST nodes (if the comparing sequences of trees is considered, the process is O (N^4). Thus, the amount of computation becomes prohibitively large.

The way to solve this problem is to partition the sets of comparisons by ranging the subtrees with hash values. It allows the straightforward detection of exact subtree

clones. If the subtrees are hashed into M buckets, then only the trees in the same bucket need be compared. It cuts the number of comparisons by a factor of M. The factor is selected as probably the same order as N; in practice, M = 10% N means little additional space at great savings in terms of computation. It is found that the cost of comparing individual trees after partition averages close to a constant, rather than O(N), and so hashing allows this computation to occur in practice in time O(N).

```
Clones = Ø
For each subtree i
    If mass(i)>=MassThreshold
    Then subtree be hashed into bucket
For each subtree i and j in the same bucket
    If Compare_Tree(i, j) > Similarity_Threshold
    Then { for each subtree s of i
             if Is_Member(clones, s)
             then Remove_Clone_Pair(clones, s)
           for each subtree s of j
             if Is_Member(clones, s)
             then Remove_Clone_Pair(clones, s)
         }
```

Figure 5: Basic Subtree Clone Detection Algorithm

Instead of comparing subtrees for exact equality (Type I), they should be handled for similarity by using a few parameters. The similarity threshold parameter enables the user to define how similar two subtrees should be. The similarity between two subtrees is calculated with the following formula:

Similarity = 2 * S / (2 * S + L + R)

Where:

S = number of shared nodes

L = number of different nodes in subtree1

R = number of different nodes in subtree2

The mass threshold parameter defines the minimum subtree mass (number of nodes) value to be considered, so that small pieces of code are ignored. The algorithm is described in figure 5.

Finding Clone Sequences:

The above section described how to find subtrees, or namely single clones. But in practice, clones in the source are always large clones, which consist of a sequence of single clones, such as sequences of statements in workflow, or a sequence of declarations at the beginning of the program. Thus, the single clones need to be processed, in order to detect the complete clone in ASTs

```
void f()
{x = 0;
 a=1;
 b=2;
 w=4;
}
```

```
void g()
{y = 2;
 a=1;
 b=2;
 i=5;
 }
```

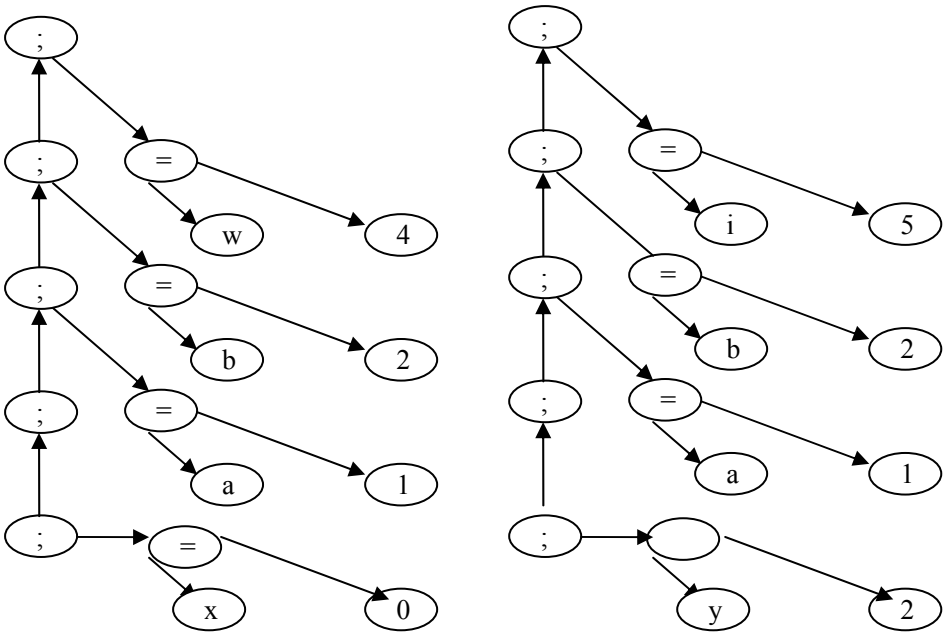

Figure 6: Example of clone sequence

The sequences of subtrees in AST are a consequence of the occurrence of rules

encapsulating sequences of zero or more syntactic constructs in the dialect grammar, they are showed up in ASTs as right- or left-leaning trees with some kind of identical sequencing operator as root, in figure 6, two segments of code will be represented in trees, which indicate the sequence of single clone.

```
Build the list structures describing sequences
For k = Minimum_Sequence_Length_Threshold
    To Maximum_Sequence_Length
Place all subsequences of length k into buckets
according to subsequence hash
For each subsequence i and j in the same bucket
    If Compare_Sequence (i, j, k) > Similarity_Threshold
    Then {Remove_Sequence_Subclones_of (clones, i, j, k)
          Add_Sequence_Clone_Pair(Clones, i, j, k)
          }
```

Figure 7: Sequence Detection Algorithm

The basic subtree clone detection algorithm is not able to detect the clone sequence. In order to find sequence clones, a list structure is to be built, where each list is associated with a sequence in the program, and stores the hash code of each subtree element of the associated sequence. The list structure allows computing the hash code of any particular subsequence very quickly. The sequence detection algorithm handles this by comparing each pair of subtrees containing sequence nodes, looking for the maximum length sequences that encompass previously detected clones. Short sequences are not interested. A minimum sequence length threshold parameter controls the minimum acceptable size of a sequence. Figure 7 is the sequence detection algorithm. It compares each pair of subtrees containing sequence nodes looking for the maximum length of possible sequencing that encompasses a clone.

Generalizing clones:

Another method is used to detect more complex near-miss clones after finding exact and near-miss clones. The method consists of visiting the parents of the already detected clones and check if the parent is either a near-miss clone.

```
Clones_To_Generalize = Clones
While Clones_To_Generalize ≠ Ø
    Remove clone (i, j) from Clones_To_Generalize
    If Compare_Clones (Parent_Of (i), Parent_Of (j)) > Similarity_Threshold
    Then {
        Remove_Clone_Pair (Clones, i, j)
        Add_Clone_Pair (Clones, Parent_Of (i), Parent_Of (j))
        Add_Clone_Pair (Clones_To_Generalize,
                        Parent_Of (i), Parent_Of (j))
        }
End While
```

Figure 8: Detecting more complex clones

A significant advantage of this method is that any near-miss clones must be assembled from some set of exact sub clones, and therefore no near-miss clones will be missed. The detected clone set is the union of sequence clones and the results of the clone generalization process. After all clones were found, a macro is generated, which abstracts each pair of clones.

2.2 Clone Detection Tool – CCDIML

The Bauhaus toolbox contains a tool, with which duplicated code can be found automatically. This tool is a command line – CCDIML (Code Clone Detection on IML), it uses the method by Baxter el. (ICSM 1998) and compares abstract syntax-trees, which is introduced the above section. The abstract syntax-trees in the Bauhaus are represented in form of the IML, which is introduced in chapter 1.1.

2.2.1 To be analysed system –concepts

The to be analysed system for Bauhaus project is **concepts**, a medium-sized system written in C. the author is Christian Lindig who developed it during his Ph. D. research at the University of Braunschweig. This tool is being used world wide in the reengineering community. It is fairly well designed and implemented and does not show the symptoms of typical legacy systems. In particular, its consequent naming convention is helpful for understanding the system even if you are neither familiar with the system itself nor with its application domain.

The source files of this system are contained in two directories that form two separate layers. Directory *lib* is a library of general abstract data types, namely, a hash table, a set, and a list. These abstract data types may be reused for other systems. Directory *src* contains the application-specific code.

The size of the source files of concepts are summarized in the following table:

Directory	#file type	Lines of code
Lib	4 header files (.h)	256
	4 C files (.c)	1.494
Src	11 header files (.h)	447
	10 C files (.c)	5758
	29 files	7955

2.2.2 Usage of CCDIML

The usage of ccdiml is following:

ccdiml [-only_routines] [-all_statements] [-pre_minlines <number>] [-pre_mindepth <number>] [-pre_minweight <number>] [-minlines <number>] [-mindepth <number>] [-minweight <number>] [-type3gap <number>] [-outformat <param>] [-version -verbose -trace -usage -help] <inputfile>

Where:

-only_routines	consider only Routines
-all_statements	consider all statements in Statement_Sequences
-pre_minlines <number>	min. length during hashing phase (default 2)
-pre_mindepth <number>	min. subtree depth during hashing phase (default 1)
-pre_minweight <number>	min. subtree weight during hashing phase (default 1)
-minlines <number>	min. length for output (default 6)
-mindepth <number>	min. subtree depth for output (default 1)
-minweight <number>	min. subtree weight for output (default 1)
-type3gap <number>	max. gap in leaf nodes for type 3 clones (default 10)
-outformat <param>	output file format (either cpf or emacs; default: cpf)
-version	prints version information
-verbose	enables verbose error/diagnostic output
-trace	enables trace diagnostics
-usage or -help	prints usage information

Possible output file formats are:

cpf	Clone Pair Format as used in cloneval
emacs	for use with interactive Emacs buffers

Figure 9: Usage of CCDIML

CCDIML is a command line tool. As input file, the IML graph is used in order to detect duplicated code. According to the method of Baxter, it traverses the IML graph using its syntactic edges and hashes the subtrees into buckets during traversal. A filter can be applied to put only subtrees of a certain size into the buckets. Currently there are different filters (to define the MassThreshold**)** on line length (*-pre_minlines*), subtree depth (*-pre_mindepth*) and subtree weight (*-pre_minweight*). If you want to include all single statements directly under a Statement_Sequence node, you can do this with the *-all_statements* switch. If you only want to look for completely clone routines, use the *-only_routines* switch.

For detection of Type III clones, a maximum gap could be given after *–type3gap*. This

is determined by the number of non-artificial IML leaf nodes that between the two clone pairs when looking for Type III gaps. A value of *0* means not to detect any Type III clones.

The switch *-only_routines* disables the *-all_statements*, the *-pre_** and the *-type3gap* switches. The switch *-all_statements* may possibly override the *-pre_** switches if their threshold is set higher than a single statement.

There are several attempts made to rule out clones that are part of larger clones of the same type, clones where one code fragment overlaps the other and duplicated clones that are symmetric.

A last filter can be applied to the output: Only clones of a certain length (*-minlines*), a certain subtree depth (*-mindepth*) or a certain subtree weight (*-minweight*) could be output, which indicate the similarity in the method of Baxter.

The output is sorted and can be done in Clone Pair Format (*-outformat cpf*) as specified as "Format 1" at http://www.bauhaus-stuttgart.de/clones/ or in Emacs Clone Pair Format (-outformat emacs) which is strongly deprecated now. The Emacs mode *cpf-mode.el* can now cope with the plain Clone Pair Format, so that no special Emacs format is required anymore. The output format "emacs" may be removed in the future, so it should not be relied on.

The implementation of CCDIML is using the method of Baxter and following the flowchart of the algorithms step by step, which are introduced in the previous section. The flowchart of CCDIML is illustrated in figure 10:

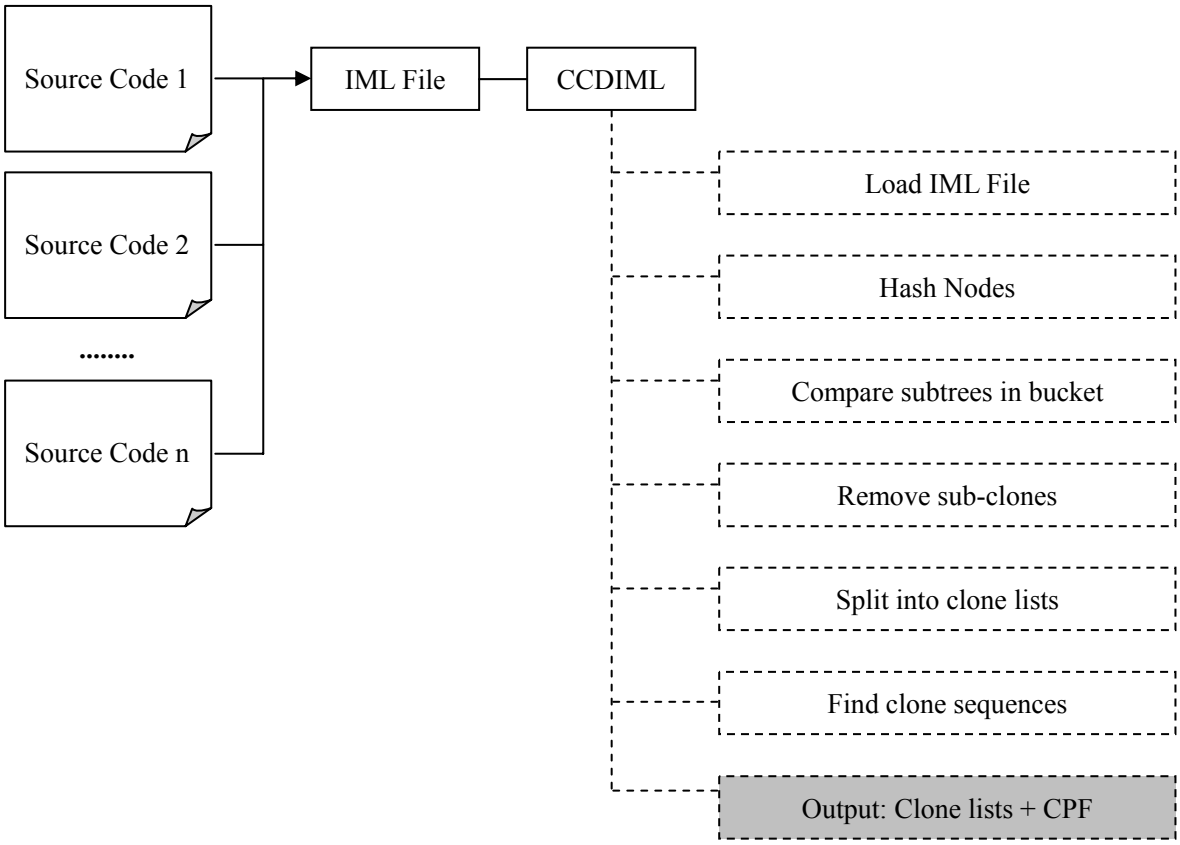

Figure 10: Flowchart of CCDIML

The left parts are source code of the system. All the source codes will be compiled into an IML file with the compiler *cafe,* which is developed in Bauhaus to compile source code into IML structure, instead of the normal C compiler. After compiling and linking through *imllink*, the generated IML file will be as the given input parameter for the tool CCDIML.

CCDIML opens the IML file first, and then hashes the IML nodes in the IML Graph into buckets according to the other given options (*-pre_minlines*, *-pre_mindepth*, *-pre_minweight*) from the command line. The subtrees are compared in the bucket with the Basic Subtree Clone Detection Algorithm (figure 5), the sub clones will be removed and the result will be split into clone lists. After finding the single clone, the next step is to find out the sequence of clones according the sequence detection algorithm (figure 7). After finding the sequence of clones, all the results are stored in

a clone list, which contains the information of the clone-pair. The clone list is to be output, either directly displayed in the screen (standard output) or saved into a Text file with the extension “CPF”, which could be displayed with a specific defined emacs mode – CPF (Clone Pair Format) mode – to show the found clones. The output clone lists is formatted as follows:

Filename1	From_Line1	To_Line1	Filename1	From_Line2	To_Line2	Clone_Type

Figure 11: Format of the CPF file

In the CPF file, each line contains the information of a clone pair. It consists of 7 fields, which gives the filename of the two clone segments, describes from which line to which line they occur and what the clone type is. The CPF file is to be displayed in a special mode programmed with Emacs Lisp, which is to be introduced in the following chapter.

2.3 Modification of CCDIML

2.3.1 Motivation of the modification

The existent code clone detection tool CCDIML analysis the IML file and find out all the clones with the method of Baxter. The results of the clones are directly output to screen or a text file, which is to be displayed in Emacs with a specific Emacs mode – CPF Mode. According to the requirements of my diploma thesis, the results of the clone pairs should be further analysed and processed, in order to extract the new functions or macros from the code clones and then replace them with the call of the functions or macros.

In the runtime environment of CCDIML, the data type *clone_lists* contains the structure information of the source code; each element in the list is a pair of clone. The structure information includes the nodes (Classes), edges (Relations) of the

subtree; it is almost the complete representation of the clone segments in the source code. In the output phase of CCDIML, only the location information of the clone segments is used. CCDIML reads the file name, begin of line number and end of the line number (figure 11) from the IML file, and writes the location information of both segments along with the clone type in a line into the CPF file. The CPF file is a pure text file; it consists of many lines, which describes the location information of each clone pair.

The motivation of saving the clone list of the CCDIML is that in order to eliminate the duplicated code and replace them with new functions or macros, the whole presentation of the clone code is needed, here the subtrees are required. The subtrees of clone code would be analysed and processed later. As the case may be the requirements of function or macro, some transformation rules (code transformation will be introduced in the next chapter) are to be defined for the subtrees, we use these transformation rules to convert the discrete duplicated code into complete function- or macro-definitions, and the clones could then be replaced with the calls of the function- or macro.

As a necessary condition for my work, the content of clone list in CCDIML should not simply be discarded after outputting the location information of clone pairs, but ought to be saved somewhere for later use. The source code of CCDIML would be added some new features to save the clone lists.

2.3.2 Specification of the modification

The storage of IML file is based on the IML hierarchy. All the IML nodes have tight relationship with each other, and all the IML class are inherited from the root node – *IML_Root*. The *IML_Root* is the foundation of IML hierarchy, it is a pointer deduced from the class *Storables.Storable*, which is a primitive class for the storage of the IML data structure.

2.3.2.1 Storage of the clone pairs

In CCDIML, the data structure of clone lists is a user defined record in Ada95; this is a nesting record, which includes the subtrees of the clone pair.

As in chapter 1 introduced, the whole system can be represented as a complete IML tree, and a root node is the as a handle (entry) of the tree for further analysis and traversal. In the same way, each code segment (duplicated code) in the system, no matter what the scale is, can be also represented as a subtree with its own root node. Therefore, if a clone pair is detected, each segment in the pair could be represented with a root node, through which the subtree of duplicated code could be accessed.

As mentioned above, there are two possibilities to represent the clone segment, the first is single node, and the other is node sequence. A record for a code fragment is defined to describe the different type of clone; it has a Boolean variable to inform that the clone is single node or node sequence. In term of different clone, this record type save the root node of a single subtree or the sequence of root nodes for a sequence of subtrees, respectively. Each clone pair involves two code fragments and the type of the clone, and all the clone pairs are inserted into a list after they are detected. The clone list is to be recurrence processed finally to get the location information of each clone pair and then output them.

It is known that all the saved information (root node or sequence of root nodes) are all pointers. That means all the elements saved in clone list are all temporary values, they are only the address of each subtree in the memory. The pointers can not be simply saved into a file with its absolute value, while next time when the original IML file is loaded, the address in the memory for the IML tree will be totally reassigned, at this time when the file with the absolute address of pointer is recalled, all the saved root nodes can not correspond to the actual address at all.

2.3.2.2 New class for clone pairs

As analysed above, the pointers are relative values in memory and they cannot be directly saved into a single file. We deal with this problem with converse reflection. Since the absolute value of the pointer can not be saved because of the new assignment for loading IML file every time, we could imagine that the pointer also as relative value to be saved upon the related object, here it is the root of *IML_Graph – IML_Root*. This intuitive idea comes from that the clone list may be saved as an attachment on a node in the *IML_Graph*; and the *IML_Graph* could to be saved as an IML file actually. If the IML file is loaded some when later, then the clone list could be also be fetched and analysed, without any information lost.

In IML hierarchy, there exists a package, which can realize the above thoughts, the package is called *IML_Attributes*. In this package, the so called attachment can be seen as an attribute for each node in the *IML_Graph*. The procedure of using the attribute is described in figure 12.

The first step in figure 12 shows that it is necessary that the type to be used for the attribute is of a subclass of *Storables.Storable_Ptr*. As described above, the data structure of clone pairs in CCDIML is an embedded record. The record of clones includes the clone type and two clone fragments, which is also a record consisting of a root node or a sequence of root nodes.

The structure of record in Ada95 is obviously incompatible with the class *Storables.Storable* - the primitive storage unit in IML hierarchy. Therefore, what we shall do is to make the representation of clone pairs to accord with the class *Storables.Storable*, so that the result of the clone lists (each element of the list is a clone pair) can be directly saved as an attribute to the *IML_Graph* for the later use. The implementation of the new class will be described in the following chapter.

2.3.2.3 Storage by using attributes

It is suggested that the clone list could be an instance of the *Generic_Storable_Lists*, which is a subclass derived from Storables.Storable. It ensures that this kind of type can be saved as an attribute to the *IML_Graph*.

To use the attribute, follow the steps:

1. **Make sure that the type to be used for the attribute is of a subclass of Storables.Storable_Ptr**
2. **call the member function in the package**

```
My_Temp : IML_Attributes.Attribute
        := IML.Attributes.Register(IML_Graph=>My_Graph,
                                   Name        =>"My_Name"
```

3. **to set this attribute for a node X with "My_New_Value", call**

```
IML.Attributes.Set ( IML_Graph => My_Graph,
                     For_Node    => X
                     An_Attribute     => My_Temp
                     New_Value => My_New_Value)
```

4. **to get this attribute of a node Y, call:**

```
if IML.Attribute.Is_Set(IML_Graph =>My_Graph
                        For_Node    => X
                        An_Attribute => My_Temp) then
    Value := IML.Attributes.Get(IML_Graph =>My_Graph
                                For_Node    => X
                                An_Attribute => My_Temp)
end if:
```

Figure 12: Procedure of using IML-Attribute

During the period that the clone pairs are found and to be output, they should also be appended in the list and finally set to the root of the *IML_Graph (IML_Root).* The elements in the clone list are clone pairs; according to the definition of the

Generic_Storable_Lists, the generic part of the list consists of three entries, a *List_Item*, a *List_Item_Ptr* and an *External_Tag*. *External_Tag* is a string to describe the tag of the class; *List_Item* and *List_Item_Ptr* indicate the element and the pointer of the element respectively, which should also be derived from the primitive class – *Storables.Storable*.

As introduced in the previous paragraphs, the clone pairs are defined as embedded record in CCDIML, and it is the so called *List_Item* for the clone list. At present the problem is that how to matching the type of clone pairs with the requirement of the clone list, without breaking the existing data structure and source code of the clone detection algorithm. That is to say, the record structure needs to be converted into a new subclass inherited from *Storables.Storable*.

In normal Ada program, the basic types can be converted to each by force. But in IML hierarchy, the structure of the class has its own internal definition, it is impossible to force the conversion from a standard type in Ada to a user defined specific class.

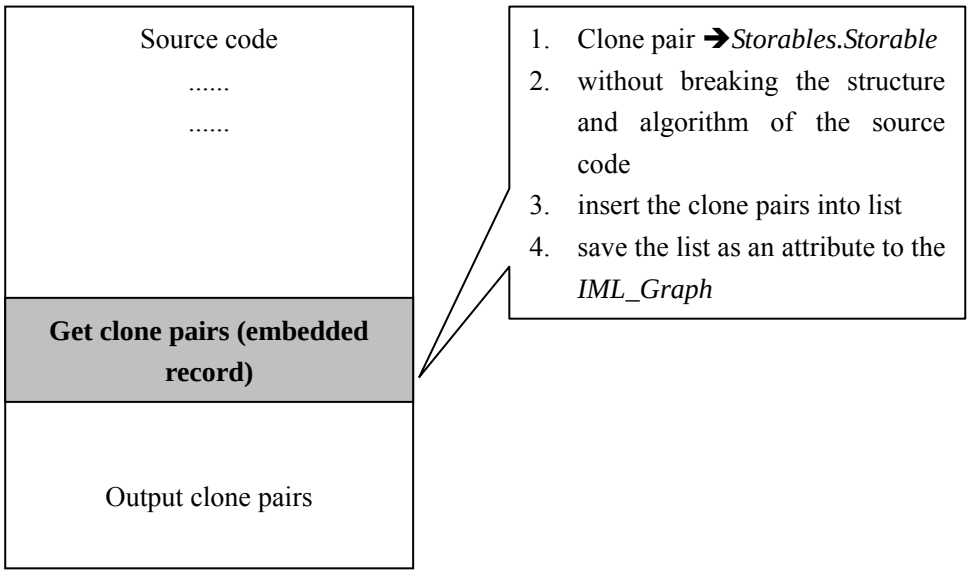

Figure 13: Modification of CCDIML

The way to solve this problem is to define a new class, which is derived from the

primitive class *Storables.Storable*, with all the in CCDIML defined information for the clones as a record, then the new class could be appended to the list. The new class should have the same name and same structure as the existing record in CCDIML, so that the usage of the clone pair in CCDIML doesn't need to be changed, to be modified is only the definition. The detail modification of CCDIML will be described in the following chapter.

Chapter 3 Code Transformation

The code transformation [REEG03] will be described in this chapter. At first the concept of code transformation is to be introduced, then some typical products or projects of code transformation. The last section is the specification of the code transformation.

3.1 Concepts of transformation

Many programming problems can be thought as transforming a single input text into a single output text, sorting a list of numbers, processing data to generate statistics, formatting text, or even compiling a program to machine code can be thought of in this way. This is the basic model of code transformation.

As a definition, a transformation is a partial function t:

a) t: specification/program → specification/program
b) examples: Compiler, YACC, program to restructure

The transformations are often represented as rewrite-rules with pattern variable. The rewrite rules (transformation rule) look like: (the bold words are key words)

```
rule eliminate_additive_identity
    Replace [expression]
        T[expression] + 0
    By
        T
end rule
```

Figure 14: Transformation Rules

In general, a transformation system is the system, which enables the semantically well-defined (partly-) mechanized modification of the program.

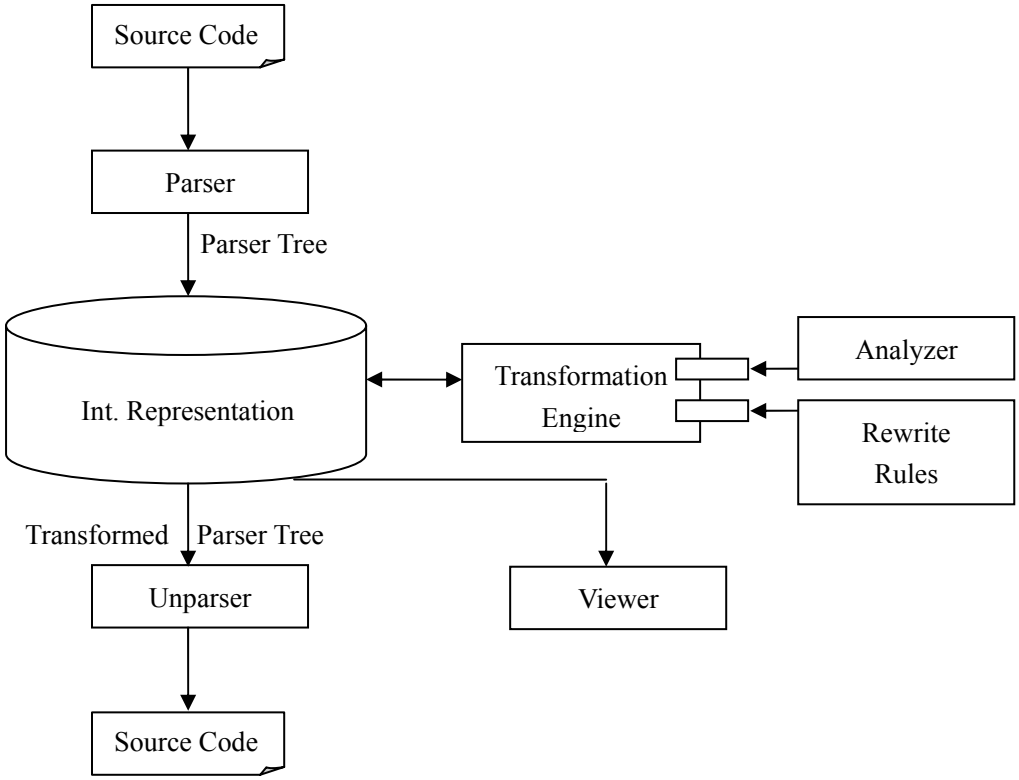

Figure 15: Transformation System

According to figure 15, the source code is to be parsed into a complete parser tree, then the tree is as input object for the transformation system, after processing the transformation, the input parser tree is restructured and a transformed parser tree is output. Finally the transformed parser tree would be unparsed into source text again, which is the expected result of the transformation. To execute a transformation, the following process should be carried out step by step:

The first step of the transformation system is to find out the location of the source code, on which the transformation rule should apply. There are two approaches to find the location of code, first is to find the predefined pattern in the source code; the second way for location is to be specified by user.

The second step is to examine the precondition of the transformation. Precondition is an expression in the transformation rules, whose function is to judge whether the transformation needs to be executed or not.

```
rule eliminate_additive_identity
    Replace [expression]
        T1[expression] + T2[expression]    -- pattern
    Where
        Value (T2) = 0                     -- precondition
    By
        T                                  -- replacement
end rule
```

Figure 16: Precondition of transformation system

The WHERE clause is a precondition for the rule. It is always knotted with the applicability of the transformation.

```
procedure eliminate_additive_identity(n: in out node) is
begin
    if type(n) = plus then
        If value (n.right) = 0 then
            -- replacement
            replace_child(parent(n), from =>n, to=>n.left);
            n.left.parent := n.parent;
            delete_tree(n.right);
            delete(n);
        end if;
    end if;
end;
```

Figure 17: Implementation of transformation rules

The last step is to realize the transformation rule in case that the precondition is fulfilled. The implementation of the transformation rules is as a procedure or function direct on the abstract syntax trees, in order to restructure it and then output. An example of implementation of transformation rule is presented in figure 17, the programming language is Ada95.

After transformation, the target code must obtains certain attributes of the source code, a few attributes should be partly changed, include performance, structure, capability of adjustment etc. In many cases (but not all cases), the semantic of the source code should remain the same.

3.2 Examples of transformation system

Two examples of transformation system will be introduced. The first is TXL, from which I got a global view of the workflow of transformation. And the other is Munich Project CIP, which shows a complete transformation system with detail specification.

3.2.1 TXL

TXL is a generic transformation system, which is developed by Queens University, Canada. It is a unique programming language specifically designed to support computer software analysis and source transformation tasks, distinctive for C, C++, Java, JavaScript, Modula, Object Pascal, and XML. It is the evolving result of more than fifteen years of concentrated research on rule-based structural transformation as a paradigm for the rapid solution of complex computing problems.

TXL was indeed originally named "Turing Helper", later changed to "**T**uring e**X**tender **L**anguage".

TXL has the following properties:

1. an abstract syntax tree is generated in the transformation
2. TXL is a hybrid functional / rule-based language with unification, implied iteration and deep pattern match
3. the abstract syntax tree is traversed and the transformation rules is also applied automatically, as long as it is possible;

Each TXL program has two components:

- A description of the structures to be transformed, specified as a directly interpreted BNF grammar, in context-free ambiguous form. As in previous section introduced, the description of the structure in TXL is a parser tree based on a suitable grammar definition.
- A set of structural transformation tules, or rewrite rules. The rules are the kernel of the transformation. It tells the transform engine what to do and how to do it.

The TXL programming language is unique in that it is has a pure functional superstructure that provides scoping, abstraction, parameterization and recursion, over Prolog-like structural rewriting rules providing pattern search, unification and implicit iteration. The formal semantics and implementation of TXL are based on formal tree rewriting, but the trees are largely hidden from the user due to the by-example style of rule specification. The abstract syntax tree is a intermediate presentation for the incoming source code.

TXL is best at tasks that involve structural analysis and transformation of formal notations such as programming languages, specification languages, structured documents and so on. It has been widely used in research applications in industry and academia as well as in production commercial applications handling inputs of up to 100,000 source lines per input file.

3.2.2 The Munich Project CIP

This section introduces the project CIP – Computer-Aided Intuition-Guided Programming – at the Technical University of Munich. The central theme of this project is program development by transformation, a methodology which is felt to become more and more important.

The work of CIP originated from two rather different motivations: on the one hand, it is to be seen as an attempt to gain methodical experience with non-toy, medium-size software projects and in this way, to demonstrate the feasibility of the CIP approach as a software engineering discipline. On the other hand, the system is intended to incorporate recent ideas as well as experience with the own prototype system and other people's systems. Thus, in the very end, it is to constitute the basis for a practicable software development tool usable by other people either in gaining experience themselves or in producing software.

Within the CIP project a prototype transformation system has been developed according to the specific CIP view of transformational programming. Starting from a formal (pre-algorithmic) specification based on suitable algebraic types, the development passed half a dozen intermediate versions and ended in a Pascal program running under CMS on a Siemens 7.865 computer and also under UNIX on a Micro-VAX at the Technical University of Munich. The purpose of the prototype system is the interactive, transformational manipulation of program schemes.

The goal requirements of CIP project are the following:

- Portable and adaptable to different hardware configurations and operating systems with reasonable effort
- The user environment of the system should be as comfortable as possible
- The core of the system should be independent of a particular language
- The system should be extensible

The prototype has been used for experiments by its creators, within students' programming courses, and to verify parts of its own development. Its main role, however, has been to serve as the major software tool in the development of the definitive transformation system CIP-S.

3.3 Specification of code transformation

This section specifies the requirements of the transformation for the duplicated code. The emphasis of the specification is placed on the transformation rules, which analyze the found clone subtree from the *IML_Graph* and convert the subtree to user selected output mode, either function or macro. The detail implementation of the transformation will be described in the following chapters.

3.3.1 Background of transformation

As mentioned in above chapters and sections, the duplicated code could be detected per CCDIML and the result of the clone could be output to standard output device or a text file. In order to keep the encapsulation of the program, the clone code should be not only found out, but also to be eliminated. The approach to eliminate the duplicated code is to generate public functions or macros, and then replace the clone with the call of the correlative function or macro.

As requirement of my work, the transformation rules for duplicated code (Type-1 and Type-2 of clones) should to be created. As an alternative for the user, the duplicated code should be replaced with macro- or function calls. The realization for the definition of the macros and functions must be based on an analysis of the abstract syntax trees, which contains all the necessary information of the represented code. The output of this analysis is the definitions of macro- or function with suitable formal parameters as well as that of the macro- or function call, which is to be used to replace the duplicated code, with the appropriate actual parameters. The definition of functions/macros and the calls of them are saved in two separate text file. The

definition file is a normal *C* file, which could be referenced by the others files in the program. The file of function- macro call is a specified text file; it contains all the calls of function/macro for the clone pairs, each line in the text file is a call of function/macro for one clone segment. The calls in the file are saved in the same order as the clone pairs in the CPF file.

To design the transformation for the code clone replacement, the steps of the above introduced transformation system should be followed, including the phase of creating a parser tree, processing transformation with pre-defined transformation rules, and creating target code by unparsing the transformed syntax tree.

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Original	**x**		**=**		**M**	**A**	**X**		**+**		**a**	
in IML	**x**		**=**		**1**		**+**		**a**			
generated	**x**		**=**		**1**		**+**		*****	a		
Result	**x**		**=**		**M**	**A**	**X**		*****	+		a

According to the requirements of my work, the transformation should be realized from text to text. The process of text to text transformation is as follows: first, the source code is to be represented in IML file, in which the macro and comment information have been lost. Then the IML file is to be analyzed to get necessary position information in the code segment, which should be inserted or replaced in the target code. The position information is used to be cooperated with the original text to generate new functions/macros for replacement. It is theoretically a good idea but in practice very difficult to be implemented. Above table is an example, the generated function could contain some additional symbols (e.g. "*") to fulfill the definition of syntax or grammar of the program language. The position of * before *a* is detected to be in column 9 in the IML file, but this number is not accord with the actual position of *a* in the original code. If it is inserted, the result can not be correct. Since the

collection of the position information from the IML file is a redundant step, the exact automatic transformation for the clone code is impossible with the IML file, so we need to choose another way to realize the semi-automatic transformation.

As described above, I take on the method to analyze the IML file and create new functions/macros from the IML nodes directly according some predefined transformation rules. The generated functions/macros are also not exact the same as the original code, but they could be later manually modified in the step of clone replacement, which is to be specified in the next chapter.

As required in the description of the task in chapter 1.3, the traditional steps of the transformation system for the code clone replacement would be little modified to fit some properties of IML structure in the Bauhaus project.

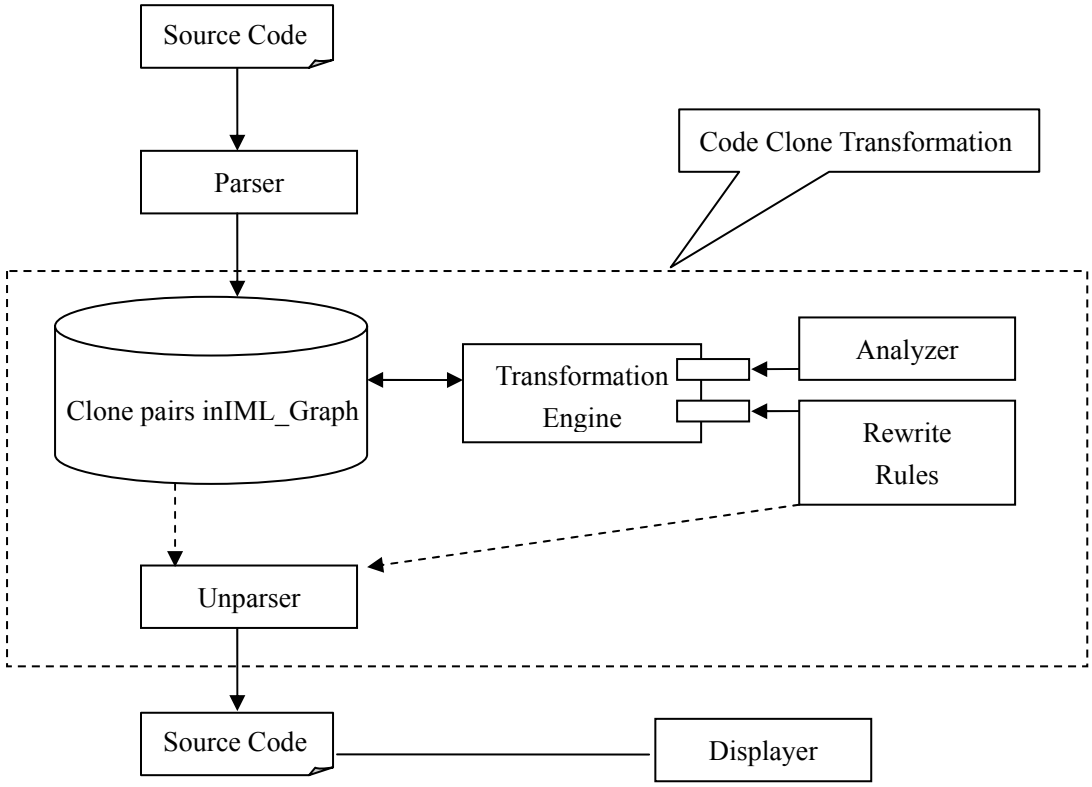

Figure 18: Transformation for Code Clone Replacement

The frame of transformation for code clone replacement is approximately the same as

the model introduced in the previous section. For the special requirements of my task, the flowchart of the transformation is partly modified, as shown in figure 18.

3.3.2 Specification of Transformation

3.3.2.1 Specification of the Workflow

Figure 18 shows the modified workflow of the transformation for the code clone replacement. The first step is the same as the traditional transformation system that the source code is parsed into a parser tree, and this abstract syntax tree will be an input parameter for the transformation engine. In Bauhaus project, the parser is cafe (C Analyzer front end), which analyses the source code and generates the abstract syntax tree with the format of IML. IML file represents the source code in all its details. As a precondition, the source code to be analyzed should be correct C file according to the standards ANSI-C 89 or ISO-C, respectively.

The second step of transformation is to be modified in my work. According to the traditional transformation system, the transformation engine processes the input abstract syntax tree and follows the rewrite rules to modify the tree, and the restructured tree will be output to the next part, to be unparsed to target code. In this paper, the second and third part – transformation engine and unparser – are combined as one step. The new step accepts the parsed AST as input parameter, and then analyses the AST to find out the needed information (as in the previous chapter described, the needed information for the transformation is the clone list, which is attached as an attribute of the *IML_Graph*); the engine fetches the clone pairs from the clone list, creating new function or macro for each pair according to the transformation rules. Finally, the definitions and calls of the new function/macro should be written into two separate text file. That is to say, the abstract syntax tree will be directly analyzed and processed into text instead of being restructured to another tree for further unparsing.

The reason for the combination of the transformation and unparser is that the program text cannot be recovered original-faithfully from the abstract syntax tree, in particular if macros and comments are used. Before cafe analyses the source code, some parts of the source code are already be handled by the preprocessor, and some information will be certainly lost, e.g. the usage of macro, which will be directly expanded in the source code by preprocessor, and the definition of macro is not stored after the processing. Therefore, we do not restructure the IML to a new one, but only analyze it and create the target code directly.

3.3.2.2 Specification of the transformation rules

The task of the transformation is to define the transformation rules for the code clone replacement. According to the requirement, the clone code should be replaced with either function or macro. The process of the transformation is specified in figure 19.
The tool for the transformation is called CCR (Code Clone Replacement), which is also a command line tool. CCR opens the IML file, in which the clone list is saved as an attribute in the IML_Root, fetches the clone list from the IML_Graph, traverses the list and analyzes each clone pair to create a new function/macro for them according to the transformation rules.

The specification of the transformation process is described based on figure 19:

- The IML file is opened, the *IML_Graph* is then loaded in memory
- The clone list, which is saved as an attribute on the node *IML_Root*, is fetched through the Bauhaus package *IML_Attributes*
- Iterate on each element of the clone list – clone pair, as long as the list is not empty, send the current clone pair to the transformation engine, where the clone pair would be analyzed and processed
- In the transformation engine, judge if the clone pair is in type III. If yes, the pair should not be processed, because we are only interested in the clones of type I and type II according to the description of the task. If not, keep on processing.

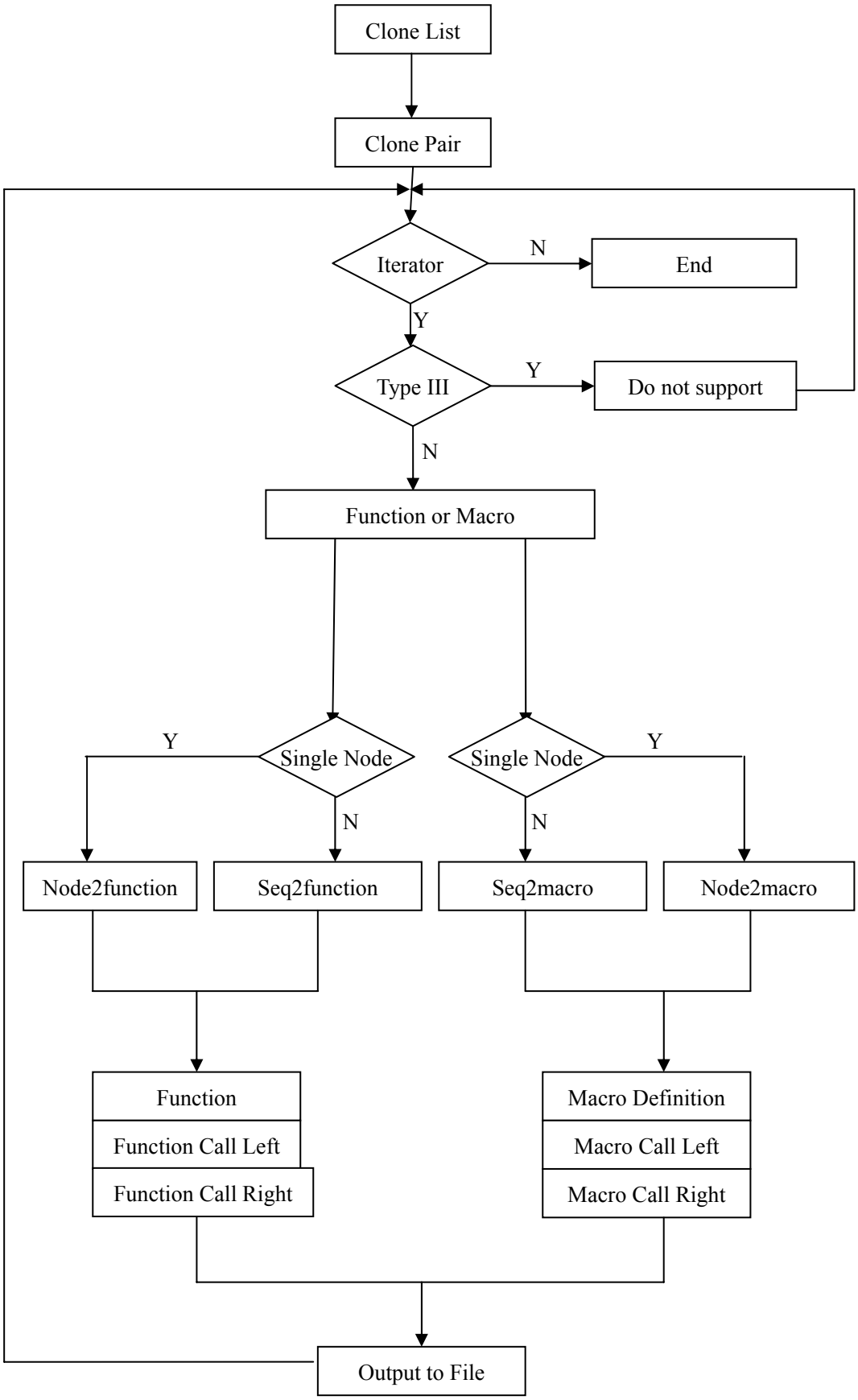

Figure 19: Work flow of transformation

- The user can choose the manner of transformation that the clone code should be replaced with functions or macros. Then user may select the form of transformation is in the command line to set the option of the parameter. As designed for the input parameter for CCR, if user wants to do the replacement with functions, the parameter *–function* should be give in the command line; otherwise, the parameter *–macro* should be given if the user wants to replace with macros. The options *–function* and *–macro* are mutually exclusive to each other. That is to say, in the same time, only one parameter can be given. If no parameter is given, the default manner of the transformation is to replace the clone code with functions
- According to the given parameter, the transformation engine passes the clone pair to the relevant module – function transformation module and macro transformation module, respectively
- In both modules, the clone pair would be traversed, analyzed and processed in the following steps:
- To judge if the clone fragment is a single node or a sequence of clone
- If single node, send it to module – *Node2function (Node2macro)*
- If sequence of node, send it to module – *Seq2function (Seq2macro)*
- The results of transformation consist of three parts: a new function- macro definition, a function- macro call for the left clone code, and a function- macro call for the right clone code. The function- macro call may contain different actual parameters for each clone code (left and right)
- All the definitions of the new function- macro is to be saved in a C file, which can be displayed in Emacs mode; and all the calls of function- macro is to be saved in another specific file with the extension *.ccr*, which could be used to replace the clone code in Emacs mode
- Iteratively fetch the next clone pair in the clone list, until the list is empty

The above mentioned process of the transformation can be summarized into a formal description – Transformation rule. The transformation rules in this paper are just

conceptual expression for the implementation of the tool CCR. They describe the transformation process in a formal way, which make the process precise. The format of the transformation rules uses the definition of Txl language (introduced in 3.2.1)

Transformation Rule 1: transform the clone pair to defined unit (function/macro)
rule Clone_To_Unit **replace** Node Clone_Fragment **where** user option (-function or -macro) from command line **by** new functions or macros (invoke the following rules) **end rule**

Transformation Rule 2: transform single node to function
rule Node_To_Function **replace** Node Clone_Fragment.Node **where** Node is the root of the subtree **by** new function: 1. traverse node to get all parameter 2. create function head with parameter list – f(t1 v1, t2, v2…) 3. unparse node to get the function body 4. assemble the head and body to generate a full function **end rule**

Transformation Rule 3: transform node sequence to function

rule Node_Sequence_To_Function

replace Node_Sequence

Clone_Segment.Node_From … Clone_Segment.Node_To

where

the clone code consists of a sequence of subtrees, each subtree has a root node, the node is ordered in sequence, from Node_From to Node_To

by

new function: 1. loop process the sequence, from Node_From to Node_to

2. traverse each node to get the sectional parameters

3. unparse each node to get the sectional result of target code

4. assemble all parameters to create the function head

5. assemble all sectional result to create function body

6. assemble the head and body to generate a full function

end rule

Transformation Rule 4: transform single node to macro

rule Node_To_Macro

replace Node

Clone_Segment.Node

where

Node is the root of the subtree

by

new macro: 1. traverse node to get all parameter

2. create function head with parameter list – m(v1, v2, v3…), variable without data type

3. unparse node to get the macro body

4. assemble the head and body to generate a full macro

end rule

Transformation Rule 5: transform the node sequence to macro
rule Node_Sequence_To_Macro **replace** Node_Sequence Clone_Segment.Node_From … Clone_Segment.Node_To **where** the clone code consists of a sequence of subtrees, each subtree has a root node, the node is ordered in sequence, from Node_From to Node_To **by** new macro: 1. loop process the sequence, from Node_From to Node_to 2. traverse each node to get the sectional parameter list, without data type 3. unparse each node to get the sectional result of target code 4. assemble all parameters to create the macro head 5. assemble all sectional result to create macro body 6. assemble the head and body to generate a full macro **end rule**

The five transformation rules are the framework of the tool CCR. In order to achieve the above modules, there are also many details to be considered. A complete detailed implementation of the tool CCR will be described in the following chapters.

3.3.3 Specification of the unparsing phase

The unparsing phase seems to be the simplest step in the workflow of the code transformation. The unparser simply does an in order traverse of the IML tree, writing the leaves to the output, and gives an unparsed string to represent of the result of the transformation. But in fact, there are still something problems to complete the functions of the phase with IML.

3.3.3.1 Background of the unparser

In Bauhaus project, there exists a package, called *unparse_c_expression*, which serves to unparse the given IML node. The entry of this package is a node with the type of *Values.Value* (it is the parent for most of the expressions in the IML hierarchy), the node will be traversed in the package and the output is a string, which represents the expected target code. The package *unparse_c_expression* has built a well structured framework and implemented some class in the IML hierarchy, but it is incomplete at all, a lot of frequently used class of operators and statements are not implemented yet. By using the existing *unparse_c_expression* package, the output string is full of question marks "?", which indicates that the correlative class in the IML to be unparsed is only defined and not implemented. So it is clearly that this package needs to be completed and consummated to meet the demands of the unparsing phase of the code transformation.

On the other hand, there are still some new requirements for the package. When the clone code is to be transformed to a macro, the *unparse_c_expression* could remain the same and just unparse the IML to target code. But when the clone code should be transformed to function, the variables in the clone code need to be further handled. E.g. a piece of code

```
…..a = b + c;
      x = a + c; ……
```

is to be replaced with a function, the function should look like this:

```
…… f (T1 *a, T2 *x, T3 b, T4 c) {
        *a = b + c;
        *x = *a + c
        }
```

The reason of the "*" symbol is because of the call by reference principle. In the new function, the variables a, b, c, x are transferred as formal parameter, after the calculation in the function, the value of the formal parameter will be lost and could

not be transferred outside of the function. So the variables can not be directly transferred with its name, instead, they need to be added a "*" symbol to ensure that the call of the variable is the reference of its address, not the name, which guarantees the correctness of the calculation and the transformation.

In this paper, I simply add a "*" for all appeared parameters in the clone segments if the transformation form is function. It looks like:

```
…… f (T1 *a, T2 *x, T3 *b, T4 *c) {
                *a = *b + *c;
                *x = *a + *c
                }
```

As introduced in the previous paragraph, that the package *unparse_c_expression* has built a well structured framework and implemented some classes in the IML hierarchy. The structure of visiting the class in IML will be illustrated in figure 21 to explain the process of unparsing with the following example in figure 20.

The expression (3 + 4 * 5) could be represented in IML like:

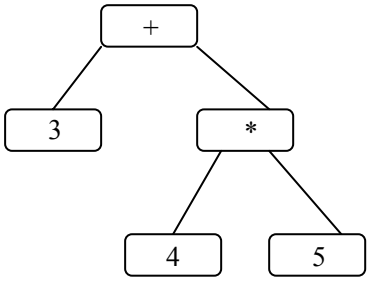

Figure 20: Example for unparsing

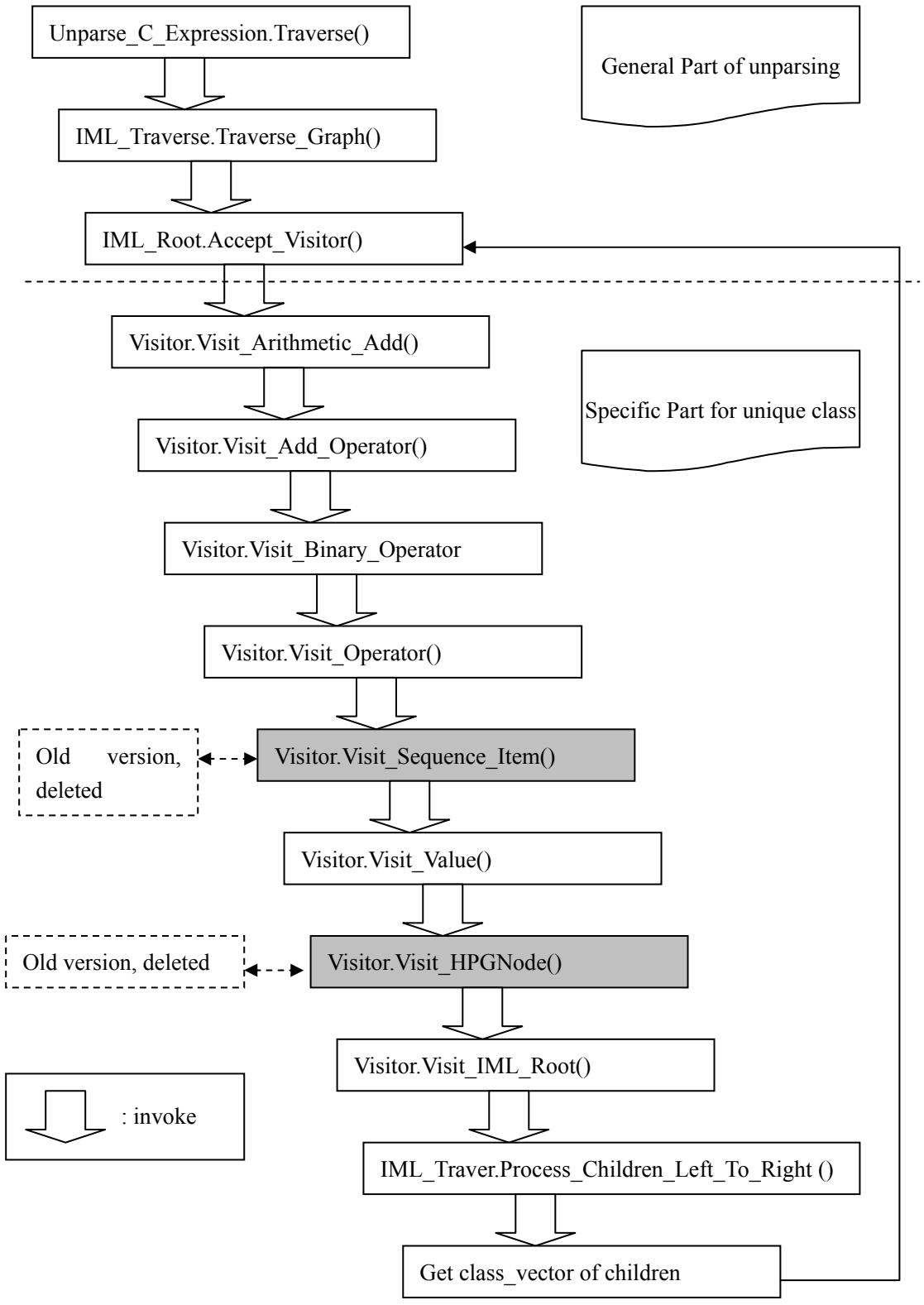

Figure 21: Process of unparsing

3.3.3.2 Process of unparsing

The expression is parsed in an IML tree (figure 20) and the root node is the add operator. As an example, we execute the unparser along the simple structured abstract syntax tree, to show the process of the unparsing phase.

The process of unparsing is (logically) divided into two parts. The first is a general part that all classes of nodes should be passed through; the second part is specific for different classes of nodes that the unparser can automatically judge what type of class is the current node and lead the unparser to the right direction according to the IML hierarchy. The flow of the two parts is to be conceptually described in the following paragraphs.

The process of the unparser is the recursive invoking on the IML hierarchy. In the general part, the entry of the package *unparse_c_expression* is the function *Unparse(),* the only input parameter is the expression with the type *Values.Value*. The function *Unparse()* calls the function *Traverse_Graph()* in the package *IML_Traversers*, and in this step, the input parameters are the expression and an appended node with the type *Visitors.Visitor*, which serves as a container to store the result of the unparser. The *Traverse_Graph()* does nothing but only calls the function *Accept_Visitor()* in the package *IML_Roots*. As in figure 1 and figure 2 illustrated, the class *IML_Roots* is an abstract class (in yellow) that some of the functions within this class should be overloaded by the derived sub class before to be used, otherwise the function could do nothing but just a empty definition.

At this moment, what the unparser has done is the general invoking from the entry to the root of the expression. From now on, the following steps are specific for different classes of nodes, which leads the unparsing phase to distinct direction according to the IML hierarchy. As described in the previous paragraph, the *IML_Roots* is an abstract class and *Accept_Visitor()* is an abstract function, it needs to be overloaded. In this

step, the IML engine could automatically find out the right class, which is inherited from the *IML_Roots*. Since then, the following steps depend on the diverse classes of nodes. In this example, the root node or the abstract syntax tree is an *Arithmetic_Add* operator, so the class *IML_Roots* can recognize the class type and turn the unparsing phase to the class *Visitors*, which handles the detail analysis of the give node.

The node of *Arithmetic_Add* is recognized by the IML engine, and the function *Visit_Arithmetic_Add()* is invoked by the previous step. The rest of the steps are executed in the package *Visitors*, thus the following description will only give the name of the function, and the package name is by default *Visitors*.

As parent of the node *Arithmetic_Add*, the function *Visit_Add_Operator()* will be called. For the same reason the function *Visit_Binary_Operator(), Visit_Operator(), Visit_Sequence_Item( )*(this function exists no more in the present version of IML, but only in the old version, see figure 1 and figure 2), *Visit_Value(), Visit_HPGNode()* (this function is also deleted in the present version of IML, it exists only in the old version, see figure 1 and figure 2), *Visit_IML_Root(),* respectively.

The *Visit_IML_Root()* invokes the function *Process_Children_Left_To_Right()* in package *IML_Traversers*, which creates a *Class_Vector* (a list type in IML) of all the children of the node, and then processes each child in the Vector recursively, until the end of the tree.

3.3.3.3 Overload the visiting function

This manner of visiting the nodes fully utilizes the IML hierarchy and guarantees that each type of node will be visited within the *IML_Graph*. The package *unparse_c_expression* uses nesting recursive relations of the nodes and overloads the functions in the package *Visitors* in order to create the target code from the abstract syntax tree.

The overloading of the functions in *Visitors* analyses the IML node one by one, to customize the case that the node could encounter. The new function has two parameters the same as the definition in Visitors, the first is a node, which is to be analyzed and unparsed, and the other is a container, which restores the result as a string. Figure 22 shows the implementation of a function, which overloads the old one from the package *Visitors*.

```
Procedure Visit_While_Loop
    (V       : access Visitor_Class;
     Node : access While_Loop.While_Loop_Class'class)
    Is
     Loop_Condition : Ada.String.Unbounded.Unbounded_String;
     Loop_Body      : Ada.String.Unbounded.Unbounded_String;
Begin
    Traverse (V, While_Loops.Get_Condition(Node), Bottom);
    Loop_Condition := V.Result;

    Traverse (V, While_Loops.Get_Loop_Body(Node), Bottom);
    Loop_body := V.Result;

    V.Result := "while ( " & Loop_Condition & " ) { ";
    V.Result := V.Result & Loop_Body;
    V.Result := V.Result & "}";
End Visit_While_Loop;
```

Figure 22: Sample of overloading functions from Visitors

According to the IML hierarchy, the class of the current node, which represents the *while* expression, named *While_Loop*. The visiting function for this class is to be overloaded in the package *unparse_c_expression* in terms of the above specification. After the general part of the process, the node *While_Loop* is recognized by the IML

engine, and the overloaded function *Visit_While_Loop()* will be automatically invoked for unparsing.

The *While_Loop* is inherited from its parent *Loop_Statement*, which is an abstract class. In the implementation of the function, the attribute *Condition* of class *While_Loop* and the inherited attribute *Loop_Body* from its father will be traversed respectively. Each time after an attribute has been traversed, the intermediate result is stored in a string, and finally the intermediate results and the keyword “while” should be assembled in terms of the C program to generate the valid target code.

This is an example for the specification of the unparser; the detailed implementation will be described in the following chapter.

This package is developed just for C program; similarly we can develop some packages for other program languages (Java, Ada, etc) the same way.

Chapter 4 Clone Replacement

After code transformation, the duplicated code has been transformed and saved in two pure text files. One is a C file, in which are the definitions of functions or macros; another is a text file with the extension *.ccr*, which contains the calls of function or macro for the duplicated code. According to the requirement of the task, the duplicated clone should be replaced with the call of new function/macro to keep the encapsulation of the software. We achieve the replacement of clones in Emacs programmed with Lisp. This chapter describes the way, how to display the duplicated code and how to replace them.

4.1 Introduction of Emacs and Lisp

4.1.1 Emacs editor

The name Emacs was originally chosen as an abbreviation of Editor MACroS.

Emacs is an extensible, customizable, self-documenting real-time display editor. It was first designed for UNIX, but it is now available for windows as well. Since it is "programmable" (in a version of Lisp), it is being continuously extended to provide support for editing different types of files (HTML, TeX, C++, shell-scripting, Perl, etc.). It can also handle mail, news, spell-checking, etc. It provides interfaces for many UNIX commands as well (to mention only one: a convenient way to compare two files, based on diff). Most of these add-ons come as separate packages (which are plain-text files), that are loaded when needed.

GNU Emacs is "free software"; this means that everyone is free to use it and free to redistribute it on certain conditions. Originally Emacs actions (e.g., save file) were

accessible through various key combinations. By now, you can do quite a bit using only the mouse. However, using a key combination is quicker than "aiming and clicking" the mouse.

One other distinction between emacs and vi is that emacs allows you to edit several files at once. The window for emacs can be divided into several windows, each of which contains a view into a buffer. Each buffer typically corresponds to a different file. Many of the commands listed below are for reading files into new buffers and moving between buffers.

4.1.2 Lisp

Elisp (Emacs Lisp) is the language used to extend Emacs, the customizable text editor of choice. It is a fully functional lisp interpreter, it includes also complete buffer/text manipulation and sub process manipulation.

Lisp was first developed in the late 1950s at the Massachusetts Institute of technology for research in artificial intelligence. The great power of the Lisp language makes it superior for other purposes as well, such as writing editor commands and integrated environments.

GNU Emacs Lisp is largely inspired by Maclisp, which was written at MIT in the 1960s. It is somewhat inspired by Common Lisp, which became a standard in the 1980s. However, Emacs Lisp is much simpler than Common Lisp. (The standard Emacs distribution contains an optional extensions file, `cl.el', that adds many Common Lisp features to Emacs Lisp.)

Every Lisp expression returns some value. Every Lisp procedure is syntactically a function; when called, it returns some data object as its value. By imperative we mean that some Lisp expressions and procedures have side effects, such as storing into variables or array positions. Thus Lisp procedures are not always functions in the "pure" sense of logicians, but in practice they are frequently referred to as "functions"

anyway, even those that may have side effects, in order to emphasize that a computed result is always returned. Imperative features are usually used sparingly; while it is possible to transliterate, say, FORTRAN code directly into Lisp, the result would not exhibit typical Lisp programming style.

4.2 CPF mode in Emacs

As introduced in chapter 2, the tool CCDIML can detect the duplicated code in the system and output the clone pairs into a pure text file with the extension of *.cpf*, which indicates Clone Pair Format. Each line of the text file contains a clone pair and the clone type, and each segment in the clone pair is represented in three fields: *filename* of the duplicated code, *from* (line number) indicates begin of the segment and *to* (line number) indicates the end of the segment.

In order to display the found clones, a special emacs mode – CPF mode – is developed in Bauhaus. The CPF mode is just for the visualization and developed with Emacs Lisp. (See 4.1.2)

Before using the CPF mode, it should be installed. The CPF mode is a program file with the extension *.el* (indicates Emacs Lisp). To install this mode, the program file should be copied into some directory (the directory is referred as CPF_MODE_DIR) and put the following three lines into the ~/.emacs file:

```
(add-to-list ' load-path "CPF_MODE_DIR")

(autoload ' cpf-mode "cpf-mode")

(setq auto-mode-alist (cons ' ("\\.cpf\\'" . cpf-mode) auto-mode-alist))
```

The three lines serve in the configuration file of emacs as a trigger. When a text file with the extension *.cpf* is opened, the CPF mode will be automatically loaded for

displaying the duplicated code. These lines are also written with Emacs Lisp, they could be directly interpreted by Emacs, without being compiled.

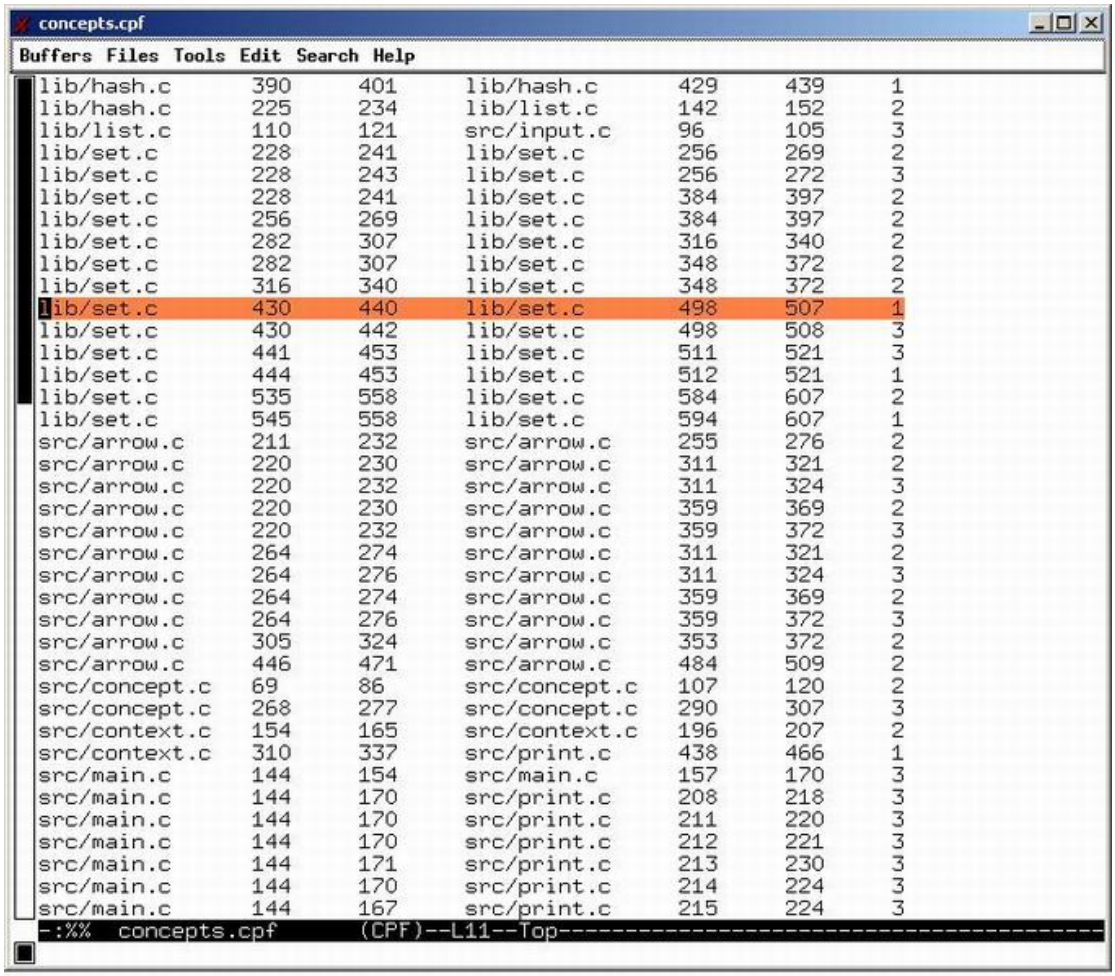

Figure 23: CPF navigator

In Emacs editor, load a file in Clone Pair Format with extension *.cpf* as usual (C-x, C-f) and the CPF mode gets entered. Moving the cursor in the CPF navigator and pressing "RET" creates a new window for the clone pairs in the line.

For the first time RET is pressed, the *base path* of the source code should interactively be given by the user, that is because that the filenames in the CPF file are relative filenames, without any information of its absolute position, so the base path parameter could tell that in which directory to find these files, otherwise the clone view frame can display nothing but just a empty window with two empty sub windows. If the

filenames in the CPF file are filenames with absolute path, the *base path* could be ignored. In case of that the wrong *base path* is entered, it could be changed in CPF mode for the current buffer with C-c C-b.

There are different colors for different typse of duplicated code in the CPF mode. As defined, Red indicates the Type I clone, Blue indicates the Type II clone and Green indicates the Type 3 clone, as shown in figure 24:

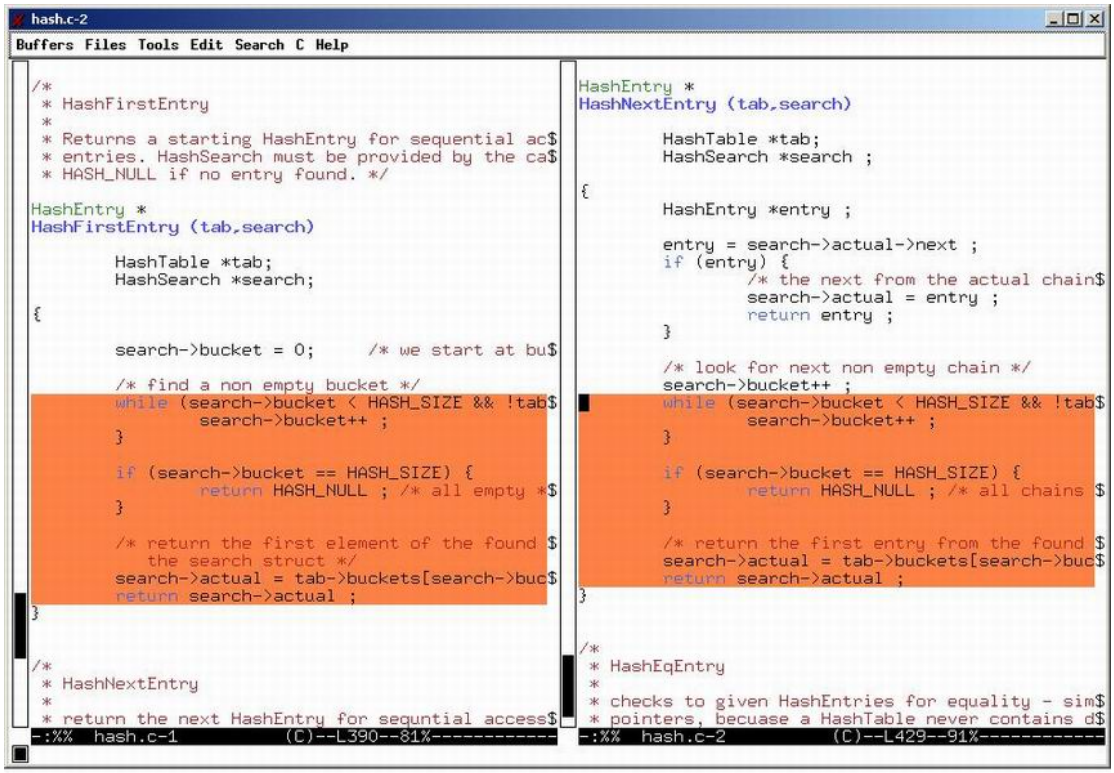

Figure 24: Clone view frame with Type I clone

The buffer in the CPF mode is read-only, it can no be modified during the time that the duplicated text are displayed.

More than one CPF file could be loaded by Emacs but only one clone pair could be displayed at one time and this is marked in the clone navigator buffer as well.

4.3 Specification to enhance the CPF mode

CPF mode realizes displaying the duplicated text in Emacs. According to the

requirement of the task, this mode should be partly modified and enhanced, in order to show not only a clone pair, but also the new functions/macros generated from the code transformation, and replace the clone pair with the calls of the functions/macros interactively.

Another reason for the enhancement of the CPF mode is that the advantages of Emacs editor could be directly utilized to meet the demands of the replacement, that the user could have the possibility of examining, adopting, modifying and rejecting individual transformations; in particular the possibility is given by Emacs, that reasonable name for the new created macros and functions to be assigned (editable); Emacs can also record all decisions of the user and allows user to cancel individual transformations (undo function)

4.3.1 Specification of the work flow for replacement

According to the requirements of the task, we design a reasonable workflow for the clone replacement based on the existing CPF mode. The flowchart is illustrated in figure 25.The original frame of CPF mode is reused. At first, the CPF file is opened in Emacs and the CPF mode will be automatically loaded. The main window is the clone navigator, which displays the CPF file like a normal text file.

In the old version of CPF mode, there is only one parameter - *base path* of the source code. The base path is to assign the absolute position (directory) for the relative filenames in the *.cpf* file. Here we add two input parameters for the clone replacement, one is the C file, which includes all the definitions of the function/macro from code transformation, this file is to be displayed in the clone view frame the same time with the clone pair; the other is the *.ccr* file, which includes all the calls of the defined function/macro. This file will not be displayed in any window but only be read into a buffer, and serves to replace the duplicated file with a user defined hot key. The two files should be exactly

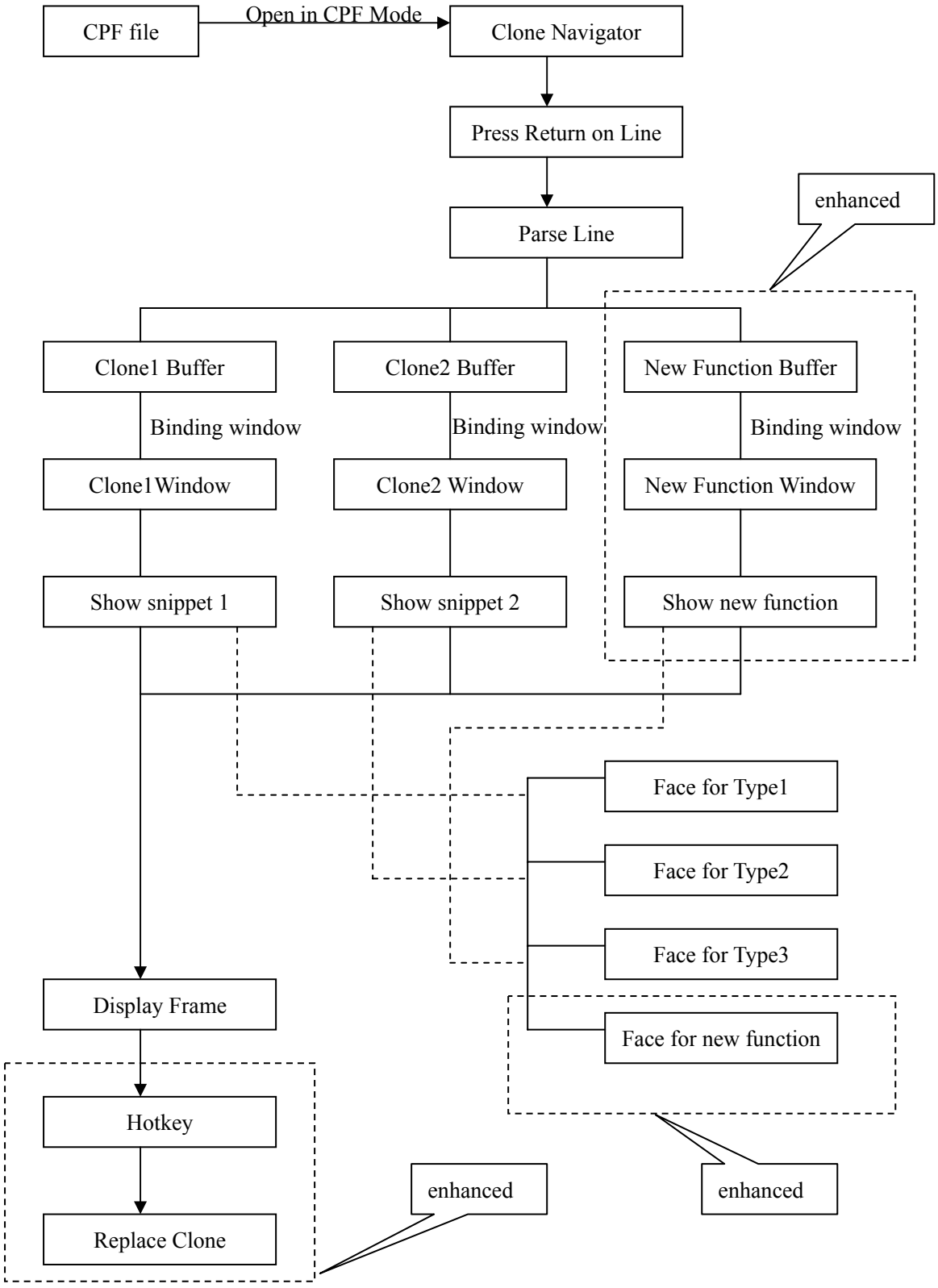

Figure 25: Flowchart of clone replacement

input for the later using, otherwise nothing could be displayed in clone view frame or

no calls could be used to replace the duplicated code. Emacs will show an error message if they are not loaded correctly.

Each time the user presses RET on a line, the content of the line will be parsed and the result of the parsing is to be sent to some other functions for displaying the duplicated code in a new frame – clone view frame. The result of the parsing is the information of the clone pair and the type of clone. They are: clone1 (filename, from, to), clone2 (filename, from, to) and a clone type. Here *from* and *to* indicate the begin and end line number in the *filename*.

After parsing the line, the next step is displaying the clone pair and the new function/macro with the parsed results. As shown in figure 24, both clone files are displayed in the clone view frame and the clone parts are highlight marked with different color according to its type of clone. In my Diploma work, I needed to add a new sub window in the frame to display the definition of the new functions/macros, which could be edited for replacement.

To display a file with the highlighted snippet (clone segment or the C file with appropriate function) is divided into three steps: first is to create a buffer and a window (in fact it is a sub window in the clone view frame) for the given file, then bind the buffer and window to make them work in cooperate with each other, at last invoke a function to find the position of the clone part (function definition) in the file and mark them with highlight. The color of the definition of the functions/macros should be different from the clone type (Red, Blue and Green), so we need to define a new face for the definition of the functions/macros.

The user can move the cursor in clone navigator to select the line of the clone, and press RETS to display them. As required in my work, a new hotkey should be defined to replace the duplicated code with the call of new function/macro. As all clone pairs in the CPF file are represented with their location information (*from* line-nr *to* line-nr), after replacement, the segment of clone (*from-to* lines) is substituted with the call of a

new function/macro, which holds only one line. So the location information of the relevant clone pairs should be adjusted to accommodate the changing in the source code. The adjustment after the clone replacement is miscellaneous because of the nesting or overlapping relationship of different types of clone, every possibility should be considered to insure the accurate calculation for replacement. The specification of the clone replacement will be described in the following section.

4.3.2 Specification of locations adjustment

As introduced in the previous chapter, there are three types of code clone: Type I, Type II and Type III. Type I indicates the same duplicated code, Type II indicates the copied code with renaming of some variables, and Type III indicates the copied code with further modification. According to the definition of clone types, it is obvious that the duplicated code with different types could be overlapped or embedded with each other in a file. So the location information in the CPF file should be adjusted every time after a clone pair is replaced, in order to keep the correctness of the content in the clone navigator.

The location adjustment is designed as following: each clone in a line should be treated as a data module – **Triple**. Each triple is described in the format (*filename, from, to*), and each line in the clone navigator contains two triples and a clone type. When a clone pair is replaced with the call of function/macro (by pressing hot key), then the filed *from* and *to* of the both triples in the line should be changed. The segment of the duplicated code is then deleted and the call of function/macro is inserted, the field *from* remains the same and the field *to* is changed with the same value of *from*, because only the call (one line) is inserted and the location of the clone segment is now a new triple (filename, from, to=from). In case that one clone pair is replaced, all the clone pairs, which are concerned with the changed clone pair (in the same file), should also be adjusted immediately, otherwise the present location information in the clone navigator is false after the replacement of a clone pair.

The adjustment seems to be simply with the given description, but in fact it is more complex in varied cases. The miscellaneous relations among all kinds of clone are illustrated in the following section.

4.3.2.1 Scene 1 of clone adjustment

In the first scene, the clone pair is two segments in different files. After replacing one segment in a file, all the other clone segments for these two files should be adjusted to keep the correct information of source location. The adjustment for clone segment in the file can be analyzed with the following possibilities: (in each sub-figure, the shadowed part of the source code indicates the replaced clone segment; the attached shadowed part indicates another clone segment in the same file, which is to be adjusted). There are all together 10 cases in this scene.

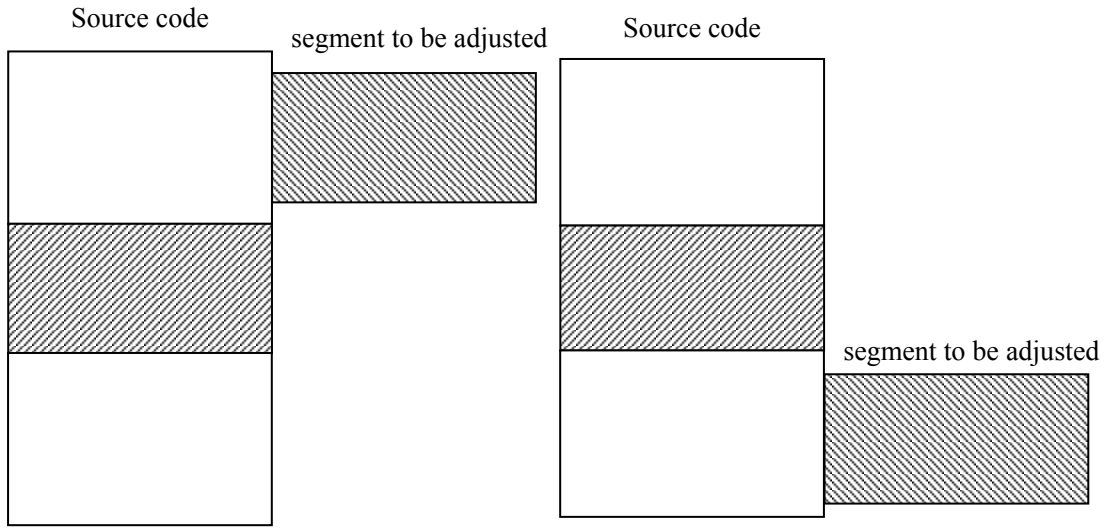

Figure 26: Clone Adjustment Scene 1-1

Figure 26 shows the case 1 of clone adjustment, the simplest case of all. The clone segment is either before the replaced segment or after it. If it is before the clone, this segment does not need to do any adjustment; if after, the field *from* and *to* just need to be reduced with the length of the clone segment (*clone-to – clone-from*).

Figure 27 shows the possibilities of embedded clones. The replaced clone segment can be a part of another segment, which is also a clone with a different. Vice versa, some other segment can also be a part of the replaced clone segment. If the clone segment is larger, then the to be adjusted segment just set it's *from* and *to* value with the same as the clone segment; if smaller, the field *from* remains the same, and the field *to* is to reduced with the length of the clone segment.

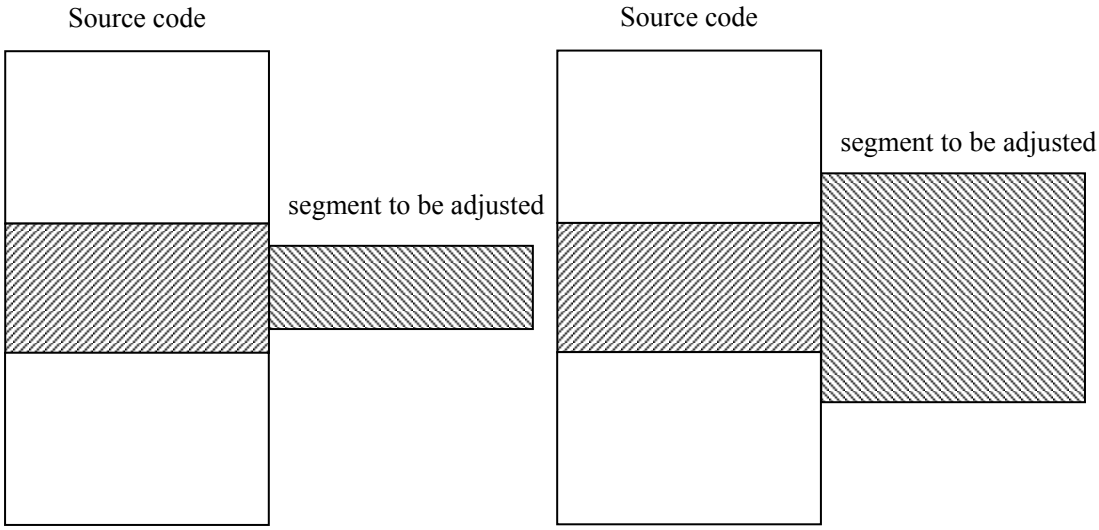

Figure 27: Clone Adjustment Scene 1-2

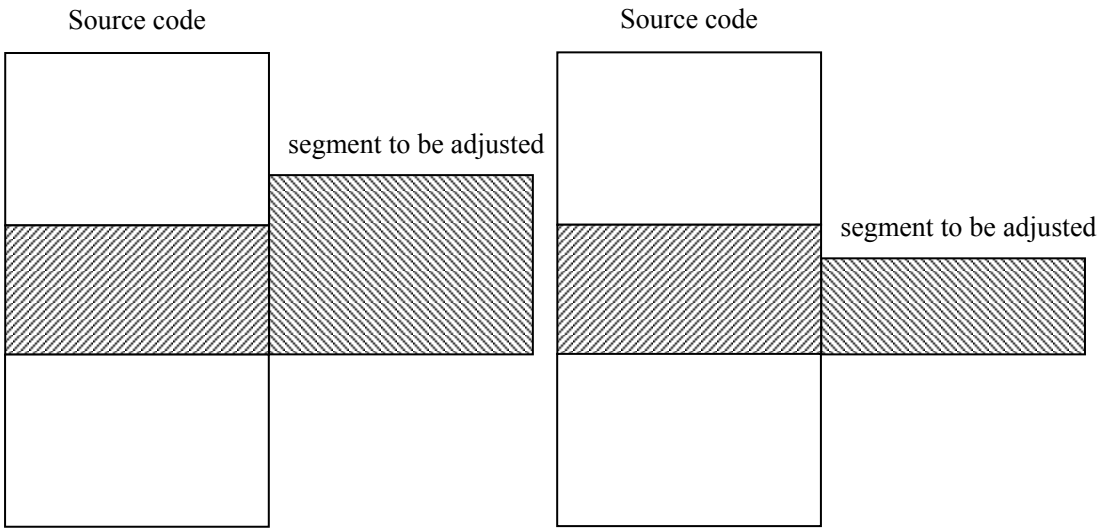

Figure 28: Clone Adjustment Scene 1-3

Figure 28 shows that the clone segment and another segment have the same value *to*, but different values *from*. In case of the left part, the field *from* remains unchanged

and the field *to* is to reduced with the length of the clone segment; in case of the right part, the segment is set to the same value as the clone segment because it is totally embedded in the clone segment.

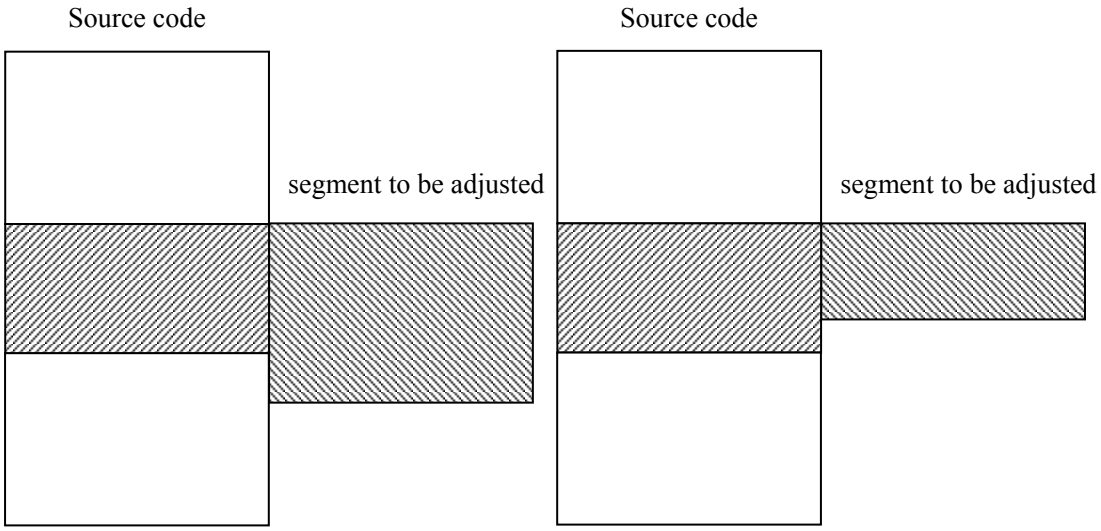

Figure 29: Clone Adjustment Scene 1-4

Figure 29 shows the inverse case of figure 28. The clone segment and another segment have the save value *from*, but different value *to*. The analysis for this case is similar as in figure 28.

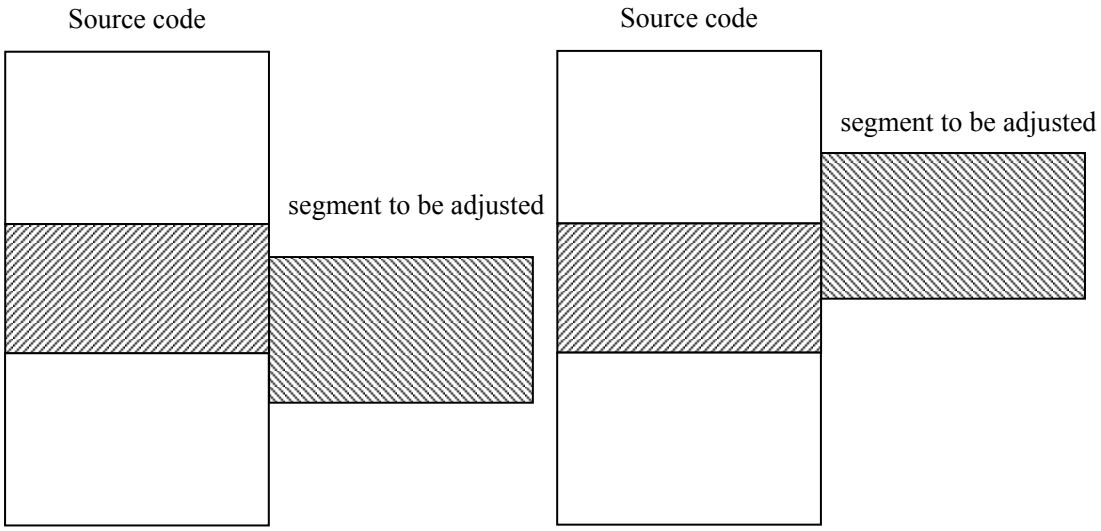

Figure 30: Clone Adjustment Scene 1-5

Figure 30 shows the case that the clone segment can be partly overlapped with another segment, which is unlikely except among some Type III clone segments. The

process of the clone adjustment can refer to the case in figure 27, 28 and 29.

In the left case of figure 30, the *from* value of the segment is set to the same as the value of *clone-from*, and the *to* value is set to the *(from + (to – clone-to))*. In the right case of figure 30, the *from* value of the segment remains the same and the *to* value is set to the same as the value of *clone-from*.

In fact, many clone segments are not code clones any more after the adjustment. The adjustment is just to keep the correctness of the location information for the clone pairs in the CPF navigator for further displaying.

4.3.2.2 Scene 2 of clone adjustment

Another scene of the clone adjustment is that the two clone segments in the clone pair are in the same file. The clone adjustment in this scene can be analyzed similarly as in the first scene, but it can be more complex. The following figures show all the cases within this scene. There are all together 19 cases in this scene.

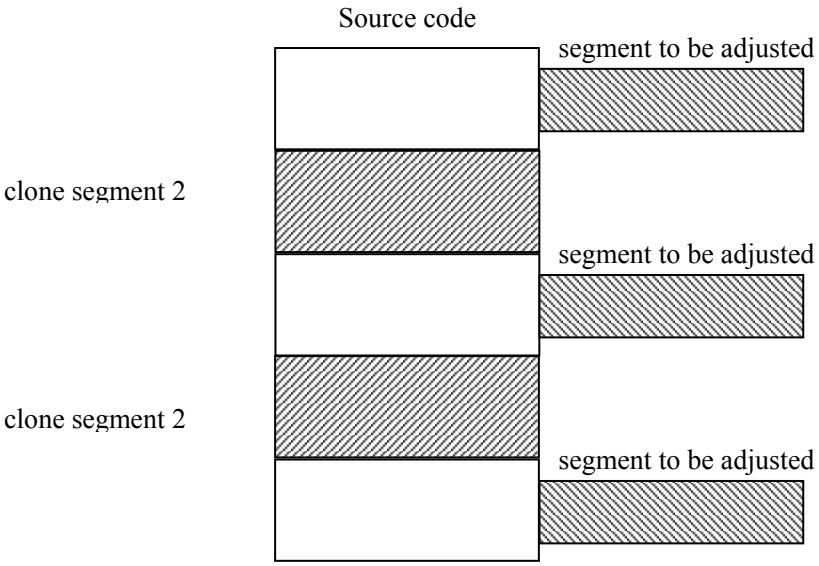

Figure 31: Clone Adjustment Scene 2-1

Figure 31 shows the simplest cases that the segment, which is to be adjusted, is either before the lower clone segment, between the two segments, or after the higher clone segment. The adjustment for these cases could refer to the computation introduced for figure 26.

Figure 32 shows the embedded cases. In this scene, there are four possibilities of the embedded segment, and in the first scene only two. Each case should be considered and calculated. The adjustment for these cases could refer to the computation introduced for figure 27.

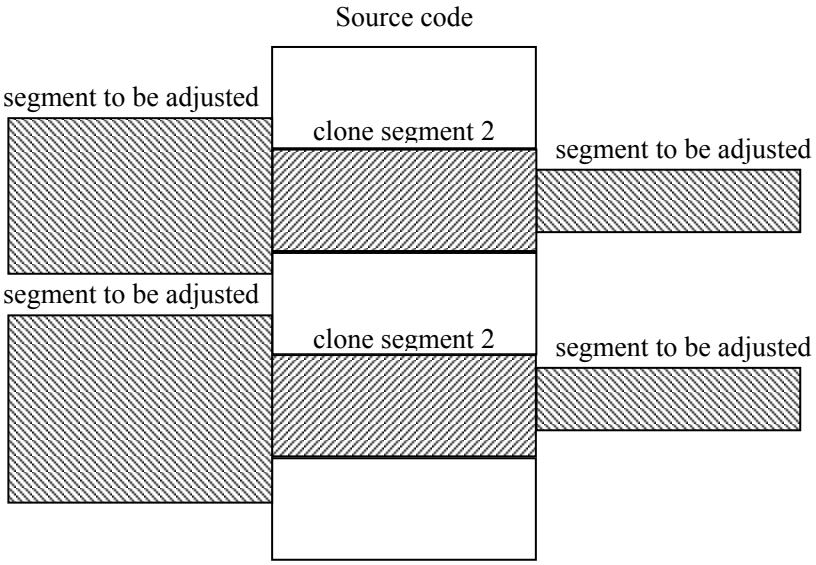

Figure 32: Clone Adjustment Scene 2-2

Figure 33 shows the case that the segment has the same value *to* as the clone segment, but different value *from*. The adjustment here is similar with the previous analysis for figure 28.

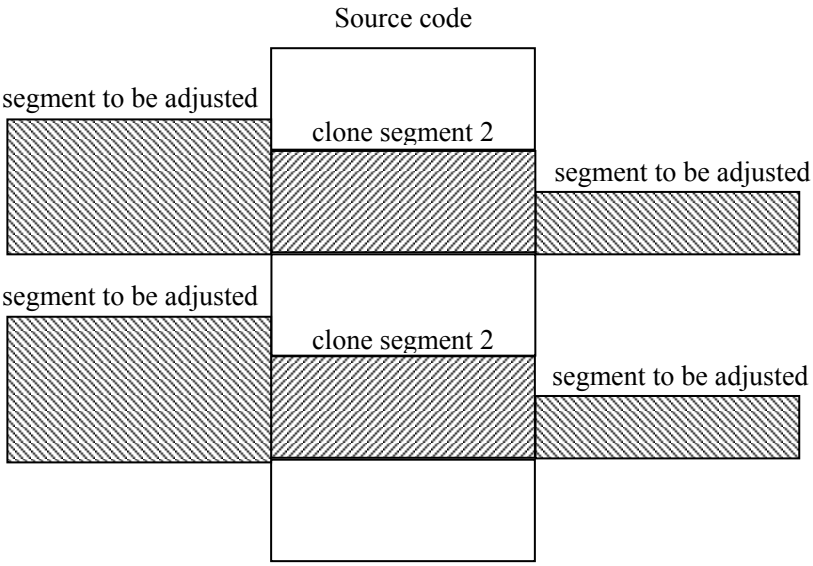

Figure 33: Clone Adjustment Scene 2-3

Figure 34 shows the case that the segment has the same value *from* as the clone segment, but different value *to*. The adjustment here is similar with the previous analysis for figure 29.

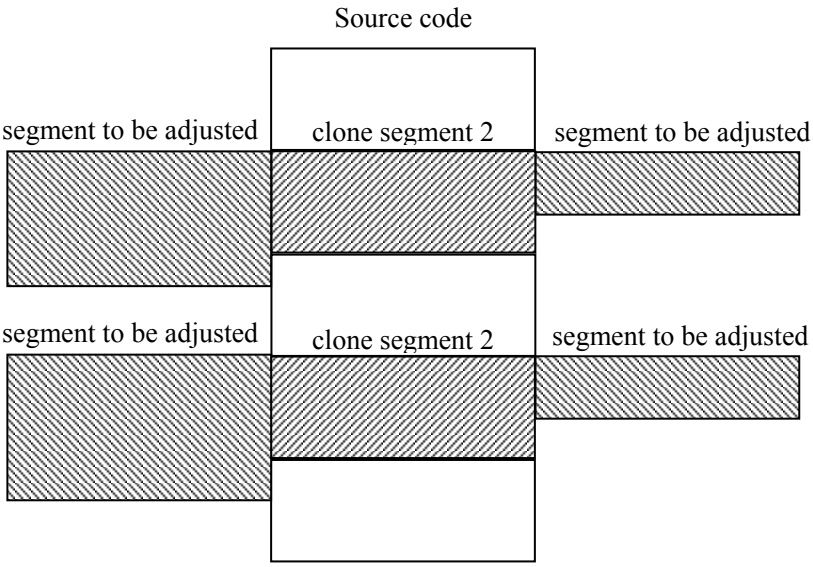

Figure 34: Clone Adjustment Scene 2-4

Figure 35 shows the case that the segment is partly overlapped with the clone segment, which is also unlikely except among some Type III clones. The adjustment here is similar with the previous analysis for figure 30.

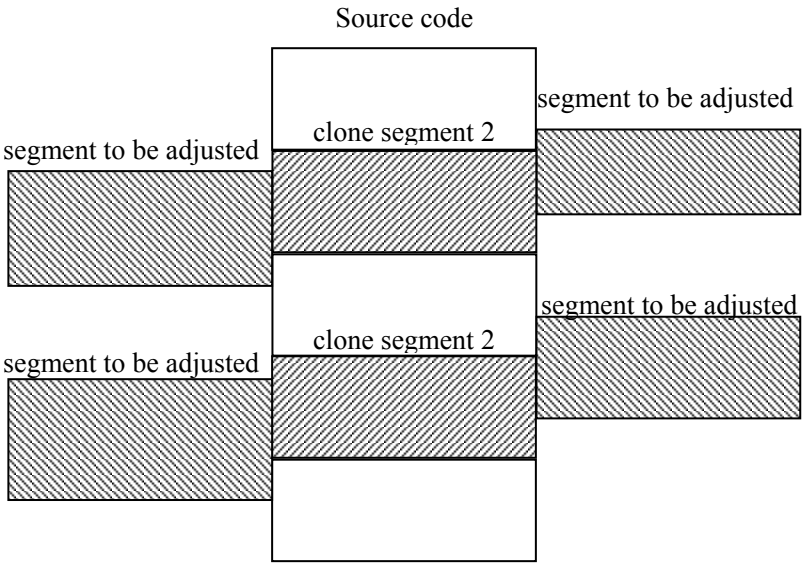

Figure 35: Clone Adjustment Scene 2-5

Chapter 5 Implementation

According to the requirements and the specifications described in chapter 2, 3 and 4, the implementation of the task consists of three parts. The first part is to extend the existing code clone detection tool – CCDIML - in order to save the clone list for further analysis. The second part is the main work of this Diploma thesis; it is to transform the clone segments from the clone list into functions/macros in term of the user's request. The last is to provide an interactive interface to display the functions/macros and to replace the clone pair with the call of these functions/macros.

5.1 Overview of implementation

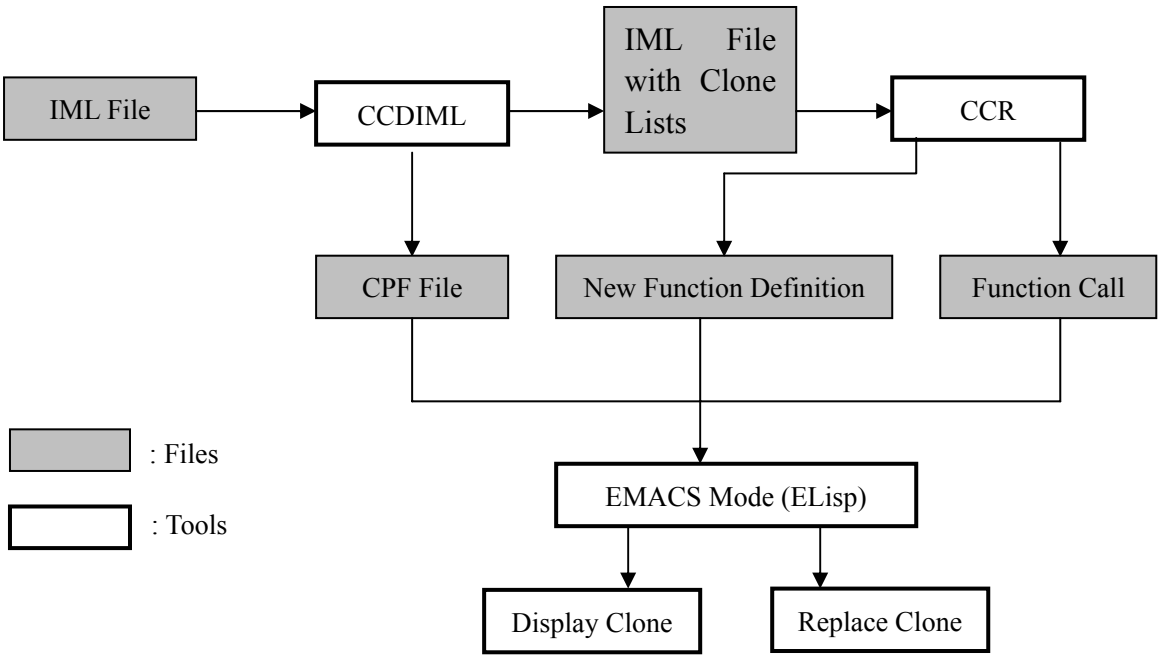

Figure 36: Structure of implementation

The implementation could be summarized in figure 36. The programming language in the first and second part is Ada95 with Gnat compiler, and in the third part it is Emacs Lisp. The platform for the total task is Solaris.

5.2 Modification of CCDIML

The main modification of the CCDIML is to create a new class for the clone pair, which is to be saved in a list. The reason for the new class is that the clone pair is to be inserted into clone list as an element, and the clone list is to be saved as an attribute in the *IML_Graph*. As defined in the package *IML_Attributes*, the value, which is to be saved on a node, must be the type *Storables.Storable*. The clone list can be inherited from the type *Generic_Storable_Lists*, which is also derived from the primitive class *Storables.Storable*. And according to the definition of the clone list, each element should be in the class of *Storables.Storable*. As in the source code of CCDIML, the clone pair is defined as an embedded record type. This type is obviously not compatible with the IML class *Storables.Storable*.

The storage of the clone pair needs to be modified, but the structure and the algorithms of the program should remain the same. That is to say, the new storage of the clone pair should be developed with two characters: first is that the clone pair is to be compatible with *Storables.Storable*; the second is that the new storage should have the same structure as the original definition of clone pair. As analyzed above, the new class is designed in figure 37.

As a new class is defined, some of the member functions must be overloaded to implement the procedure that each time the new class can be read and written correctly. As shown in figure 38, the functions are *Load_Node(), Save_Node() and Mark_Node().* The overloaded functions indicate the fixed sequences of the variables, types and sub classes in the new class could be read and written in the same order. In the clone pair, the sequence for reading and writing is (refer to the sequence of definition in figure 37): 1. read/write Boolean value in the fragment 1, then the node or node sequence respectively according to the Boolean value; 2 read/write Boolean value in fragment 2, then the node or node sequence respectively according to the Boolean value; 3. read/write the clone type.

```
Package Clone_Pairs is
   ……
   type Code_Fragments_Record (Sequence : Boolean := False) is record
    case Sequence is
      when False =>
        Node       : Storables.Storable;
      when True =>
        Start_Node : Storables.Storable;
        End_Node   : Storables.Storable;
    end case;
  end record;

  type Code_Fragments is access all Code_Fragments_Record;

  type Code_Fragment_Pairs is array (1 .. 2) of Code_Fragments;

  type Clone_Pairs_Class is new Storables.Storable_Class with record
    Code_Fragment : Code_Fragment_Pairs;
    Clone_Type    : Clone_Types;
  end record;

  type Clone_Pairs is access all Clone_Pairs_Class'Class;

  No_Clone_Pairs : constant Clone_Pairs := null;

  function Get_Class_Tag(Self : in Clone_Pairs_Class)
    return Standard.Storables.Class_Tag;

   procedure Load_Node ( Stream : in Bauhaus_IO.In_Stream_Type;
                         Node : access Clone_Pairs_Class);

   procedure Save_Node ( Stream : in Bauhaus_IO.In_Stream_Type;
                         Node : access Clone_Pairs_Class);

   procedure Mark_Node ( Node : access Clone_Pairs_Class);
……
end Clone_Pairs;
```

Figure 37: New class for clone pair

The new class can be directly used in the source code of CCDIML, without any influence on the structure and algorithm. During the period that the clone pairs are

being output, the clone pairs should also be inserted into a clone list, whose element is the type *Clone_Pairs* – the new class inherited from *Storables.Storable*. Finally the clone list is saved as an attribute of the *IML_Graph* and the *IML_Graph* is to be saved in an IML file for the further analysis.

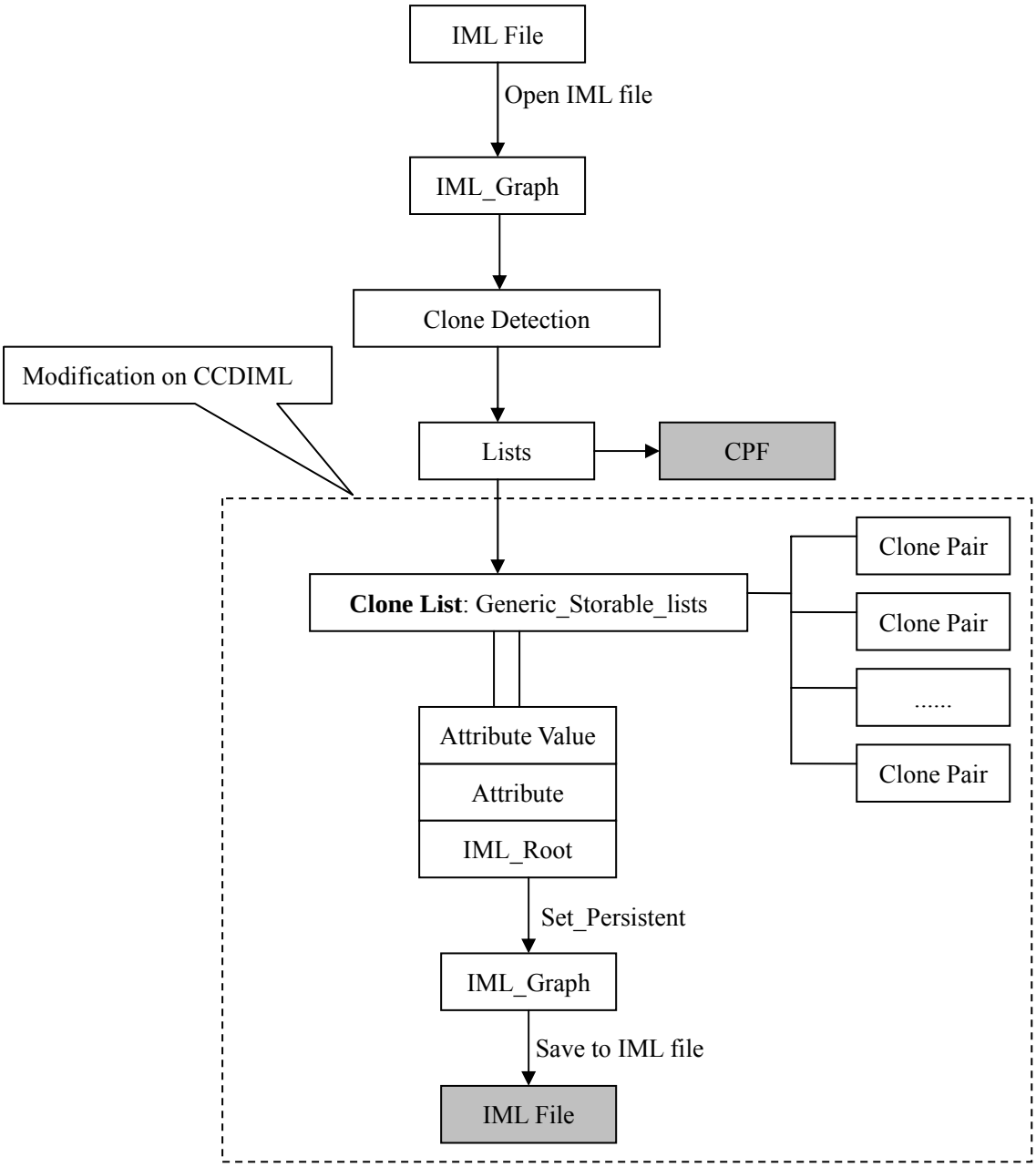

Figure 38: Flowchart of saving the clone list as attribute

The process of using the package *IML_Attributes* has been introduced in the specification in chapter 2.3.2.3.

When the *IML_Graph* is saved into an IML file, the overloaded function *Save_Node()*

in the new class *Clone_Pairs* then works. It is called by the IML engine automatically and tells how to save the content of the class in the right sequence. Similarly, when an IML file is opened, the function *Load_Node()* will be called to for the same reason.

5.3 Implementation of code transformation

I developed a tool with the name CCR (Code Clone Replacement) to realize the transformation from the duplicated code to function/macro. This is a command line tool, it analyses the IML file with the clone information as attribute in the *IML_Graph*, and processes the clone pairs according to the transformation rules, then creates new functions/macros and saves them in text files for further replacement.

5.3.1 Usage of CCR

The usage of tool CCR is illustrated in figure 39:

ccr [-function] [-macro] [-version -verbose -veryverbose -trace -usage
-license -help] <inputfile>

Where:

-function	replaced by functions
-macro	replaced by macros
-version	prints version information
-verbose	enables verbose error/diagnostic output
-veryverbose	enables very verbose error/diagnostic output (includes -verbose)
-trace	enables trace diagnostics
-license	prints license information
-usage or -help	prints usage information

Possible output file formats are:

.h	definition file of new functions/macros: for displaying
.ccr	calls of the functions/macros: for replacement

Figure 39: Usage of CCR

The IML file, which is described in chapter 5.1, is as an input file for the CCR. As defined, user can indicate that the clones in the IML file should be transformed into functions or macros with the option *–function –macro*. The two options are mutually

exclusive, that means, only one option could be given each time. If no option is given, the transformation uses *–function* as default value.

5.3.2 Code transformation

The flowchart of the code transformation is illustrated in figure 19. The implementation can be distributed in the following steps for each transformation:

- Get the clone pair from the clone list (iteratively), do transformation;

  while S_Lists.More(Iter) loop

  S_Lists.Next(Iter, Item);

  Node_Analyse(Item); -- entry of code transformation

  end loop;

- Define a list structure to store the variables for each clone segments. The list is inherited from the class *Ordered_Sets*, and each element in the set is a record to represent a variable item.

  *type **Var_Item** is record*

  Var_Name : Unbounded_String;

  Var_Type : Unbounded_String;

  end record;

  *package **Var_Sets** is new Ordered_Sets(Var_Item, "=", "<")*

  ***Var_List** : Var_Sets.Set;*

  The data type *Var_List* is used to store the variables under the root. The function of the operators "=" and "<" are overloaded in definition of the package Var_Sets, in order to carry out the comparison of the element in the list. The code for "=" and "<" is not shown here.

- Judge if a node is variable, the class *Entity_L_Value* is used. The condition of a variable node is:

  if Node.all in Entity_L_Values.Entity_L_Value_Class'CLASS

```
        then return TRUE;
      else return FALSE;
```

- When a node is recognized as variable, the name and type should be stored into the *Var_List*. The name and type of the variable can be fetched from the node:

```
function Node_Name(Node: in Storables.Storable) return String
is VAR_OC_Entity : OC_Entitys.OC_Entity;
   VAR_Identifier : Identifiers.Identifier;
begin
      Var_OC_Entity := Entity_L_Values.Get_Name(Entity_L_Values.Entity_L_Value(Node));
      Var_Identifier := OC_Entitys.Get_Mangled_Name(Var_OC_Entity);
      return Identifiers.Get_Name(Var_Identifiers);
end Node_Name;

function Node_Type(Node: in Storables.Storable) return String
is Var_OC_Entity : OC_Entitys.OC_Entity;
  Var_Type_Node: T_Nodes.T_Node;
begin
      Var_OC_Entity := Entity_L_Values.Get_Name(Entity_L_Values.Entity_L_Value(Node));
      Var_Type_Node := OC_Entitys.Get_Its_Type(Var_OC_Entity);
      return Unparse_C_Type.Unparse(Var_Type_Node)
end Node_Type;
```

- Transformation: the principle of transformation is to create a grammatical frame for the clone code (definition of function or macro). To create a grammatical frame, the variables and their types should be found to create the parameter list.

- A clone pair contains two clone segments. Since we concern only clones in Type I and Type II, Type I clone are complete the same and Type II clones have the same structure in IML with renaming of some variables. So we only need to analyze and

process either the left or right segment and create a mutual definition of function/macro. As a convention, we do the transformation on the left segment – *Clone_Segment(1).* The function names are defined from *f0* to *fn*, the number *i* from *0* to *n* is increased after each time a new function/macro is defined.

```
function Node2function(Node : in Storables.Storable) return Unbounded_String
  is    F_Def: Unbounded_String;
begin
      Traverse(Node);      -- get all variables under the Node
      F_Def := "void    "& Function_Name(Func_Name) & "( ";
      F_Def := F_Def & Get_Func_Parameter & ") {";    --function head created
      F_Def := F_Def & Unparse_C_Expression.Unparse(Values.Value(Node));
      F_Def:= F_Def & "}" & ASCII.LF;
      return F_Def;
end Node2function;
```

- To get all the variables under a node, the node need to be traversed. The function *Traverse(Node)* uses a recursive algorithm. It begins from the root of the subtree – the give Node, judges if the node is a variable. If yes, the name and the type of the variable is to be saved in a list, and the function is finished; if no, the function *Traverse()* will be called again on each child of the Root. This process runs recursively, until all the nodes under the root has been traversed. The results in the list are the variables in the clone segment. This function is relative long and the following presentation is only the formal description:

```
procedure Traverse_Node (Node : in Storables.Storable)
begin
      if Node_is_Variable(Node)
            Get_Node_Name(Node);
            Get_Node_Type(Node);
            if not Variable_ in_List(Var_Item)
                  Insert_var_to_List();
```

```
            end if;
            return
        else
            Get all children of Node;
            for each child Node_i
                Traverse_Node(Node_i);
            end for;
        end if;
    end Traverse_Node;
```

- If the clone pair is Type I, the variables in the segments are also the same and need to be analyzed only once; if it is Type II, the both segments should be seperately traversed to get their own variable list, the definition of the function/macro remains the same.

```
function Functoin_Call_Right return Unbounded_String
 is F_Call: unbounded_String;
begin
      F_Call := Function_Name(Func_Name) & "(";
      F_Call := F_Call & Get_Macro_Parameter & ")";
     return F_Call;
end function_Call_Right;
```

5.3.3 Implementation of Unparser

The unparsing phase is the last step of code transformation. As specified in chapter 3.3, there is already a frame of Unparser in Bauhaus, but it is incomplete. What I did in this phase is to enhance the functions for unparsing. The unparser is well structured that makes it not difficult to add new functions to improve the capacity of unparsing.

I have added together 13 new visiting functions in the package *unparse_c_expression*,

which enhance the unparsing phase of code transformation. The added functions achieve most of the unparsing job in the tool CCR, but still some information could not be recovered original-faithfully, in particular if macros and comments are used, while the macros and comments are processed by a preprocessor before they are sent to be compiled by Cafe. This is the reason, why the topic of my work is “Semi automatic elimination of the code clone”. The transformed functions/macros should be displayed in Emacs and be editable to meet the user’s demands.

The class *While_Loop* is illustrated in figure 22; it illustrates the approach how to unparse the conditional class. As another sample in the following, it shows the procedure of unparsing the class return with value. Through the two examples, it is easy to understand the means to unparse the IML classes are unparsed.

```
procedure Visit_Return_With_Value
      (V       : access Visitor_Class;
       Node : access Return_With_Value.Return_With_Value’class)
begin
      Traverse (V, Return_With_Value.Get_Expression(Node), Bottom);

      V.Result := “return “ & V.Result;
      V.Result := V.Result & “; ” & ASCII.LF;
end Visit_While_Loop;
```

Figure 40: Unparsing class Return_With_Value

5.4 Implementation of Clone Replacement

I choose Emacs as the platform for the clone replacement, for that the existing CPF mode could be utilized and extended. The Emacs mode for clone replacement is also called CPF mode, and some new key bindings will be added to enhance the function of this mode.

5.4.1 Implementation of the entry

In the modified CPF mode, the clone navigator remains unchanged. The first time a clone pair is to be displayed, three parameters should be given interactively by user; they are *base path, definition file of the functions/macros, and call file of the functions/macros*. Then the clone segments and the definition of the function/macro is displayed separately in a sub window in the clone view frame. Figure 41 shows that the function definition file should be given by user interactively.

Figure 41: Select definition file of function/macro

The code of the select path function is:

```
(defun cpfm-select-function-definition-file (filename)
  "select the function definition file for the clones"
  (interactive "FSelect function definition file: \n")
  (setq *F-DEFINITION* filename))
```

5.4.2 Implementation of the new sub window

To add a sub-window for displaying the new functions/macros is implemented as follows:

```
 (if *CLONE-WINDOW-VERTICALLY*
     (setq *FUNCTION-WINDOW* (split-window-vertically 17)) ;; size of sub window
     (setq *FUNCTION-WINDOW* (split-window-horizontally 27)))
 (if  (and  (not  (eq  (window-buffer *FUNCTION-WINDOW*)
                       *FUNCTION-BUFFER*))
            (buffer-live-p *FUNCTION-BUFFER*))
      (progn (select-window *FUNCTION-WINDOW*)
             (switch-to-buffer *FUNCTION-BUFFER*)))   ;; bind buffer and window
```

Before highlighting the corresponding function/macro to the clone pair, the position of the function/macro snippet should be calculated. The CPF file contains the location information of the clone pair, so it is easy for Emacs to display. But the position of the function/macro snippet in the FUCTION-BUFFER is not recorded anywhere, it should be calculated each time when a clone pair is displayed, in order to be highlighted to accord with the shown clone pair. In the definition file of function/macro, the tag "//@@" is added in front of each definition of function/macro. While "//" is seen as the comment symbol and "@@" is an infrequent symbol in comment, we choose the combination of these symbols as break tag for the definition. The denotations in the definition file look like:

```
//@@
f1( Parameter List)
{ function body }
//@@
f2 ( Parameter List)
{ function body}
```

The idea of calculating the location of each snippet is that when user presses RET on a line, the line number *n* is treated as index for the searching of corresponding function/macro. In the FUNCTION-BUFFER, the position of the *nth* and *(n+1)th* "//@@" is detected, the block between the begin and end position is the definition of function/macro for the clone pair.

```
(setq i 0)
(goto-line 0);
(move-to-column 0)
(while (< i index)
   (search-forward "//@@")
   (setq i (+ 1 i)))
 (setq f-begin (point))
```

(search-forward "//@@")

(setq f-end (point))

(setq f-begin (- f-begin 4)) ; back to begin of //@@

(setq f-end (- f-end 4)) ; back to the next begin of //@@

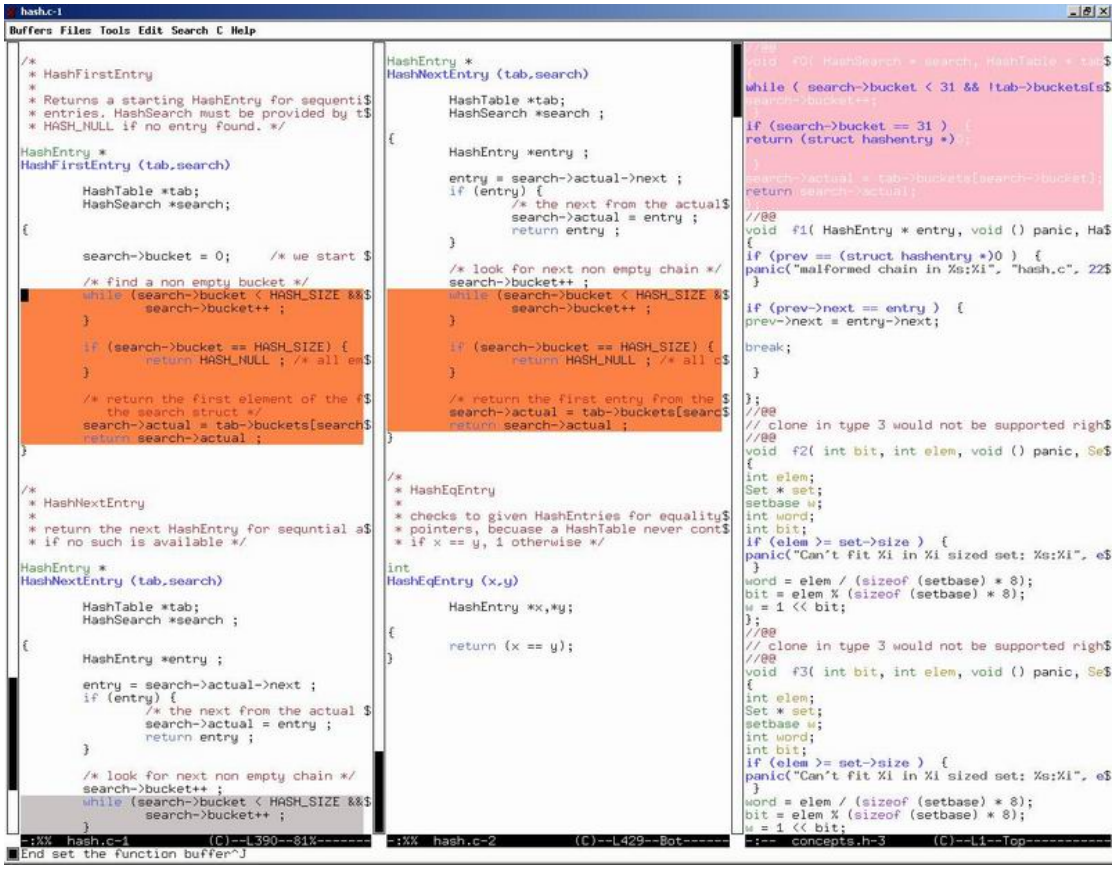

Figure 42: New clone view frame

5.4.3 Implementation of the replacement

A hot key is defined to replace a clone pair with the call of their mutual function/macro. The approach of replacement is actually an operation of delete-insert; it can be achieved in three steps. The first step is to find the location (begin, end) of each clone snippet, to remember the location and then to delete this snippet in the buffer; second, to read the call file in buffer and to find the corresponding calls for each snippet of the clone pair (left-call and right-call); the last step is to insert the calls into the clone buffer to the recorded location.

; calculate the position in each clone-window according to the from-to-line

```
(switch-to-buffer *CLONE1-BUFFER*)
(setq buffer-read-only nil)
(goto-line (+ clone1-to 1))
(move-to-column 0)
(setq end1 (point))
(goto-line clone1-from)
(setq start1 (point))

; do replace, first delete the clone region, then insert the call
 ; at the same position
(delete-region start1 end1)
(insert left-call) ……
```

After one clone pair is replaced, all the other correlative clone pair in the CPF should be adjusted to accommodate the changed location information. As analyzed in the specification, we use the triple to adjust the buffer. The adjustment of the changed location information is complex in various cases (see 4.3.2), the implementation of the adjustment is verbose and aridity, and hence, not shown here.

Chapter 6 Summary

In this chapter the experience of this diploma thesis is to be summarized in my own view. First, I will bring out several problems of the Bauhaus project, which appeared during the implementation. After that the result of the whole work is to be summed up.

6.1 Problems in Bauhaus

6.1.1 Package IML_Attributes

The package *IML_Attributes* has a full class frame in Bauhaus project, it provides a function *Set_Persistent()* to store any values with the type *Storables.Storable* as an attribute on a node in the *IML_Graph*, which is very helpful for my task. I used this function to store the clone list (see figure 38), and then save the total *IML_Graph* to an IML file. This file is to be opened in tool CCR, to get the saved clone list for the code transformation, the sequence of read and write is defined in the class *Clone_Pairs* (see 2.3).

Although I used this package in my work without any error, there is still a potential bug in the above mentioned function – *Set_Persistent()*. This bug does not have any influence for my work, but it can be a potential problem for any other program, which uses this package.

In the program CCR, the IML file with clone list as attribute is opened, and the *IML_Graph* will be further analyzed and processed.

IML_Graph := IML.IO.Load (Bauhaus.Command_Line.Input_Filename);

I write an additional line in the program after the file is opened, just to test the correctness of the I/O function of the new class *Clone_Pair_Class*:

IML.IO.Save ("test.iml", IML_Graph);

In the above procedure, the IML file is opened and directly saved into another file, the saved file should be the same as the original file. In fact, the new generated file has lost **16 bytes** by comparing with the old one. If we open and save another IML_Graph without any attributes, the problem will not appear. In despite of the lost 16 bytes, the *IML_Graph* contains the complete information of the clone list so that I can go on working. The reason of the bug is still open.

6.1.2 Result of the clone pair

The result of the clone detection has also some problems. First, the contents are different between the found clone and the saved result; see the following sample (this example is from the test file - *concepts.cpf*):

Found clone (*for loop*)	Saved Node (*if statement*)
for *(prev = *entry->bucket;; prev = prev->next) {* ***if*** *(prev == HASH_NULL) { panic(….) }* ***if*** *(prev->next == entry) {* *prev->next = entry->next; break;}* *}*	*void f1(……) {* ***if*** *(prev == HASH_NULL) { panic (…) }* ***if*** *(prev->next == entry) {* *prev->next = entry->next; break;}* *}*

The found clone in the source code is a *for loop*, but the saved node in IML is only the body of the *for loop* – here is the *if statement*. The new function/macro for the clone segments is a sequence of *if statement* according to the information in the *IML_Graph*, it is not accorded with the original code and, of course it cannot be replaced.

Another problem in the result of the clone pair is that the location of the clone pair contains some delimiter, like ")", "}", etc. it will cause an syntax error after replacement of clone; see the following sample (this example is also from the test file – *concepts.cpf*):

Found clone	New function/macro
*… **{** setbase w;*	*void f2 (……){*

int bit; *if (…..) { panic(…) }* *…..* *w = SET_ONE << bit;*	*setbase w; int bit;* *if (…..) { panic(…..) }* *……; w = SET_ONE << bit;* *}*

In this case, the found clone is at the begin of the function, and the "{" is included in the region of the clone. If the segment is replaced, the "{" is lost in the source code and needed to be added manually later. The solution of this problem is to improve the program for detecting the location of the clone pair, in order to get the accurate location for the clone.

6.2 Conclusion

This Diploma thesis designed, implemented and described a prototype in the field of code transformation for the duplicated code. It analyses the clone pairs, which were saved as attributes in the *IML_Graph*, generates new functions/macros for the clone code according to the predefined transformation rules, then displays the new functions/macros and replaces the duplicated code with the calls to them. The programming language in the work is Ada95 and Emacs Lisp.

The work is based on the code clone detection tool – CCDIML, which is implemented by using the method of Baxter. When I got the work, CCDIML could process only one input IML file. During the period of my thesis, the tool is enhanced to process more files. Since CCDIML is well structured, the result of clone list remains the same as the previous version; it has no influence on the interface of my work. CCDIML may be still improved for further requirements, if only the structure of clone list is not changed, the new defined class in my work could also be directly reused in the new version for the storage of the clone pairs.

The macro is recommended to replace the duplicated code, while the definition of

functions is related to a lot of structures and includes of extern files, which could cause trouble in compiling the generated new file. But macro can be simply expanded in the source code by preprocessor, and the structure types or function calls then do not need to be considered.

My work can be divided into three parts. First is to extend the code clone detection tool – CCDIML; then the clone list from CCDIML will be analyzed and transformed into new function/macro; the last step is to replace the duplicated code with call of the function/macro. As specified in above chapters, the correctness and precision of the replacement of clones depends on the result of the clone detection. So the work can be further enhanced in collaboration with the improvement of CCDIML.

Glossary and Abbreviations

Ada95: The Ada 95 programming language

AST: Abstract **S**yntax **T**ree

Bauhaus: The Bauhaus Project of Institute of Software Technologies, Compiler Group, University of Stuttgart

Cafe: **C** **A**nalyzer **f**ront **e**nd

CCDIML: **C**ode **C**lone **D**etection on **IML**

CCR: **C**ode **C**lone **R**eplacement

CIP: **C**omputer-Aided, **I**ntuition-Guided **P**rogramming, the project CIP at the Technical University of Munich

CPF: **C**lone **P**air **F**ormat

Emacs: **E**ditor **MAC**ro**S**, Emacs is an extensible, customizable, self-documenting real-time display editor under the GNU license

Elisp: Elisp (Emacs Lisp) is the language used to extend Emacs, the customizable text editor of choice. It is a fully functional lisp interpreter, it includes also complete buffer/text manipulation and sub process manipulation.

Function Definition: Realize a user defined concrete function object. It is the same meaning of subroutine and routine.

Function Call: Invoke a function definition with actual parameter

GNAT: GNU NYU Ada Translator, it is an Ada95 compilation system by Ada Core Technologies

GNU: GNU's Not UNIX, GNU Operating System - Free Software Foundation

HPG: Hierarchical Program Graph

IDE: Integrated Development Environment

IML: **I**nter**M**ediate **L**anguage for description of the source code in Bauhaus project

TXL: **T**uring e**X**tender **L**anguage

Bibliography

[Bauhaus] Bauhaus home page, University of Stuttgart
http://www.bauhaus-stuttgart.de

[TB473] Bauhaus members, *Tour de Bauhaus, Version 4.7.3*. University of Stuttgart, Institute of Software Technologies, Compiler Group.

[IML98] Jürgen Rohrbach, *Erweiterung und Generierung einer Zwischendarstellung für C-Programme*. Studienarbeit Nr. 1662, University of Stuttgart, Institute of Computer Science, January 1998

[IML03] Tahir Karaca, Sebastian Setzer, *Erweiterung und Generierung der Zwischendarstellung IML für C++ Programme.* Diplomarbeit Nr. 2048, University of Stuttgart, Institute of Computer Science, March 2003

[Ada-RM] Ada95 Reference Manual Website
http://www.adahome.com/rm95/

[Ada95] John Barnes, *Programming in Ada95, 2nd Ed.* Addison Wesley, 1998

[Elisp] Emacs Lisp Reference Manual Website
http://www.gnu.org/software/emacs/elisp-manual/elisp.html

[KR 88] Kernighan B.W., D.M.Ritchie. *The C Programming Language, Second Edition. Prentice-Hall,* 1988

[Baxter 98] Ira D. Baxter, A. Yahin, L. Moura, M.S.Anna, L.Bier, *Clone Detection Using Abstract Syntax Tree,* ICSM 1998

[Baker 95] Brenda Baker, *On Finding Duplication and Near-Duplication in Large Software Systems*, Working Conference on Reverse Engineering 1995, IEEE

[Partsch 90] Helmut A. *Partsch, Specification and Transformation of Programs*. Springer-Verlag, 1990

[DS 87] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *Data Structure and Algorithms*. Addison-Wesley publishing company, 1987

[REEG03] Rainer Koschke, *Script of lecture Reengineering.* ISTE, University of Stuttgart

[TXL] Txl homepage, Software Technology Laboratory, Queen's University at Kingston, Canada. http://www.txl.ca

[CIP 87] G.. Goos, J. Hartmanis, *The Munich Project CIP – Volume II: The Program Transformation System CIP-S*, Springer Verlag, 1987

Erklärung

Hiermit versichere ich, daß ich diese Arbeit selbständig verfaßt und nur die angegebenen Hilfsmittel verwendet habe.

Yidong Liu