Towards Re-engineering of Linux Kernel from Procedural to Object-Oriented

M. Tech. First Stage Report

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology

by

Nishit Desai
Roll No: 04305802

under the guidance of

Prof. R. K. Joshi

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

Acknowledgment

I hereby express my sincere thanks and gratitude towards my guide *Prof. R. K. Joshi* for their constant help, encouragement and inspiration. Meeting with him have been a constant source of ideas, and gave me motivation for this project.

Nishit Desai
IIT Bombay

Abstract

Object oriented programming has many advantages over conventional procedural programming languages for constructing highly flexible, adaptable, and extensible systems. Given many benefits of object oriented systems over conventional procedural systems and the rapidly escalating costs of maintenance of systems written in conventional languages, the migration of billions of lines of procedural code into object-oriented languages is an attractive option. This report discusses existing techniques to identify objects from procedural code, and proposes new methods for identification of objects using access graph. Rule based approach to identify constructor and to associate functions to classes is also proposed. These proposed methods are also applied on small C programs and the result are shown. Effectiveness of the proposed method w.r.t. manual object extraction is also discussed.

Contents

Chapter 1

Introduction

In the early years of software development, code was mainly written using procedural languages. Over the years, code has to be modified to satisfy new requirements. Repetitive maintenance tasks carried out with little concern for architectural design make these systems unmanageable. If the specifications are not updated with maintenance, it results into loss of valuable information. To review the architecture embedded in such code in order to increase the life span of the system, it becomes essential to re-engineer existing code. Reverse engineering can be applied to understand the existing system and reproduce the specifications and design lost over the years.

Modern software development methodologies like object oriented paradigm have numerous advantages over structured paradigm. A system with well designed object oriented architecture is more adaptable to changes. A rich collection of compositional mechanisms for class formation, object instantiation, property inheritance, polymorphism and information hiding are built into object-oriented programming languages that provide support for creating systems that exhibit a high degree of reuse. Object oriented systems have their components largely decoupled. This allows changes to be made in one component without acting the other components in the system. Codes developed using the object oriented paradigms are easier to evolve and to maintain. For existing systems to take benefit of the present technology in software development, it becomes desirable to re-engineer the existing procedural code into object oriented form. Re-engineering is further explained in the following sections.

1.1 Re-engineering

Jacobsan et al. [10] describes *re-engineering as the process of creating an abstract description of a system, reason about a change at the higher abstraction level, and then re-implement the system.* Generally, an old system is chosen for re-engineering if it is difficult to change and has high business value. This could be seen from the decision matrix [10] shown in Figure 1.1. If the system has low business value and is easy to change, then we do maintenance of the system.

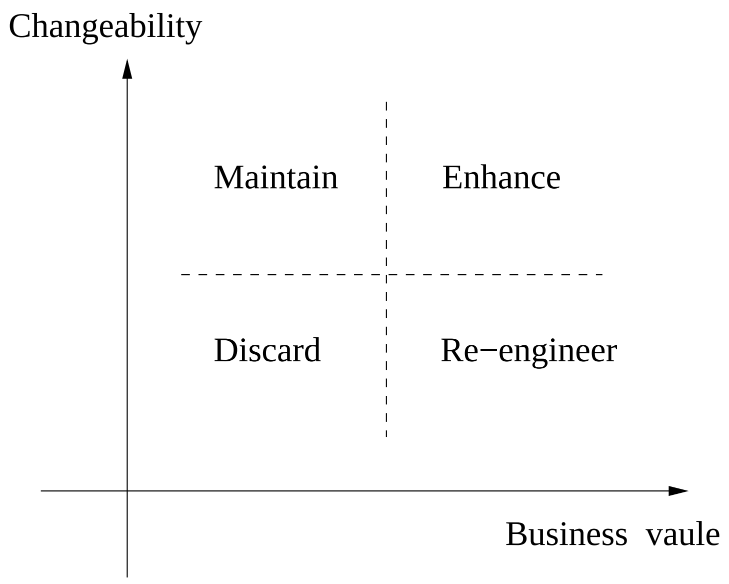

Figure 1.1: Decision matrix, what to do with old system

Re-engineering can also be expressed by following formula:

Re-engineering = Reverse engineering + △ + Forward engineering

Reverse engineering is the activity of defining a more abstract, and easier to understand, representation of the system. The goal of the reverse engineering is to capture an understanding of the behavior and the structure of the system.

Changes can be classified into two orthogonal dimensions, change of functionality and change of implementation technique. Change of functionality doesn't affect how the system is implemented. A change in implementation technique could mean moving to advanced technologies such as moving from C code to C++ code. A complete change of implementation technique will seldom occur for a large system. A change in implementation is not an easy process. The orthogonal dimensions mean that changes in implementation technique are made without changing functionality and vice versa.

Forward engineering is normal software development process.

This project is an aim to re-engineering linux kernel with a complete change in implementation technique but no change in its functionality. Linux kernel is currently implemented in procedural language C. The Change in implementation technique means converting it into Object oriented language C++.

While re-engineering procedural code into an object oriented one, we first have to do reverse-engineering and get abstraction of the system. But due to enormous complexity of code, it would be hard for the programmer to go through the entire code and detect basic classes which would be used in object oriented system. At this stage of re-engineering process, if there exists some object identification, then it will be very helpful. The term Object Identification refers to the recovery of classes from the procedural code. Thus, object identification plays a very important role in re-engineering the legacy code.

1.2 Organization Of Report

This chapter discusses the basic concepts in the field of re-engineering. The next chapter discusses existing techniques to identify objects in procedural code. Existing techniques to identify relationships between objects are discussed in chapter 3. Chapter 4 focused on the proposed method to identify abstract data objects (ADO). Chapter 5 discusses technique, proposed to identify abstract data types (ADT). Rule based approach to identify constructor and associate functions to classes is also discussed in this chapter.

Chapter 2

A Survey of Existing Techniques to Identify Objects

The Object oriented paradigm is based on object model. The major elements of this model are Object abstraction (class and instances), Encapsulation, Inheritance, Dynamic Binding. So, object model identified from procedural code should satisfy these properties. In this chapter, existing techniques to identify objects from procedural code have been discussed.

Basically, there are three types of objects those can be identified from procedural code. They are Abstract Data Type (ADT), Abstract Data Object (ADO) and Hybrid objects. The abstract data type [2] is an abstraction of a type which encapsulates all valid operations of the type and hides the details of the implementation of those operations, by providing access to instances of such a type exclusively through a well-defined set of operations. An abstract data object [7] is a group of global variables together with the routines which access them. Cross-breeding of ADTs and ADOs are called hybrid components. They consist of routines, variables, and types. An ADT is build around types wheres ADO is build around global variables. The main difference between an ADT and an ADO is that ADT can have any number of instances by declaring objects of this type whereas there is always exactly one ADO.

We use sample C code shown in Figure 2.1 to explain some of existing methods. Figure 2.1 contains files list.h, list.c, complex.c and datastructure.c. The file list.h contains declaration of structure Complex and List, file Complex.c contains operations those can be performed on structure Complex, file list.c contains operations that can perform on List and file datastructure.c contains implementation of stack and queue.

2.1 Global Based Object Identification

Liu and Wilde [12] were the first to propose global based object identification.

```
    /* ------------ list.h ------------------*/

struct Complex{
    float real;
    float imaginary;
}
struct List{
    int len;
    Complex element[100];
}

   /* ------------ list.c ------------------*/

Complex top(List l)                 -- return top element of l
Complex first(List l)               -- return first element of l
void empty(List *l)                 -- empty the list l
List remove_first(List *l)          -- remove first element of l
void remove_top(List *l)            -- remove the top element of l
void add_top(List *l, Complex c)    -- add c to top of the list

     /* ------------ complex.c ---------------*/

Complex construct(float, float)   -- construct complex from two reals
add(Complex a, Complex b)         -- add two complex number
subtract(Complex a, Complex b)    -- subtract two complex number
multiply(Complex a, Complex b)    -- multiply two complex number
divide(Complex a, Complex b)      -- Divide two complex number

     /* ------------- datastructure.c -----------*/

List stack;               -- global variable
List queue;               -- global variable
init()                    -- initialize stack and queue
void Push(Complex X)      -- push element X to the stack
Complex pop()             -- pop top element from the stack
enqueue(Complex X)        -- add element X at the end of queue
Complex dequeue()         -- remove element from front of the queue
```

Figure 2.1: Sample Code

The method work as follows

- For each global variable x, let P(x) be set of routines which directly access x.
- Consider each P(x) as a node, construct a graph G=(V,E) such that
  V = { P(x) }
  E = { P(x1)P(x2) | $P(x1) \cap P(x2) \neq \emptyset$ }
- Construct a candidate object from each strongly connected component C = (Vc,Ec) in G such that
  functions $= \cup_{P(x) \in Vc}$ P(x)
  data members $= \cup_{P(x) \in Vc}$ {x}

The graph shown in Figure 2.2 is constructed for file datastructure.c.

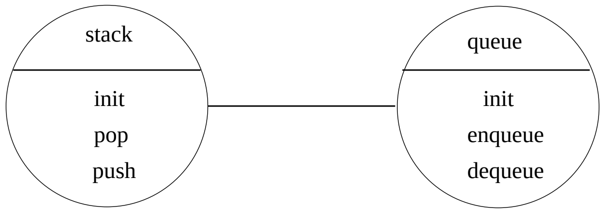

Figure 2.2: Global Based Object Identification

Dunn and Knight [4] proposed different way of constructing graph to identify objects from global variables. They propose bipartite graph, with nodes as routines and global variables. An edge is directed from a routine node to variable node if the routine accesses global variable. Again we find connected components which form object.

There are some problems in this method such as a function might be an initialization function, which access most of the global variables. The global based method would then produce only one class with all the global variables as its data members and all the functions accessing the global variables as methods of the class. For file datastructure.c of Figure 2.1, global based method would result in formation of ADO = {{stack, queue}, {init, push, pop, enqueue, dequeue}}. It is observed that stack and queue are grouped into one class, but a class for stack and class for queue is better solution. The problem is with method *init* which is accessing both global variables. Had there been no *init* method, it would have identified stack and queue as two different objects.

2.2 Type Based Object Identification

Liu and Wilde [12] also proposed object finding based on types. First a topological ordering of all types in the program is defined as follows. If a type x is used to define type y, then we say that x is Part Type of y. The Part Type relation on types is transitive.

We form graph with a node for each routine and type. Next, routines and types are connected. A type is connected to routine if it is formal parameter or return type of the routine. However, if type and its part type are connected to same routine, part type is ignored. Basic data types like int, float, char are ignored by this method. The groups consisting of connected routines and types form objects and its methods.

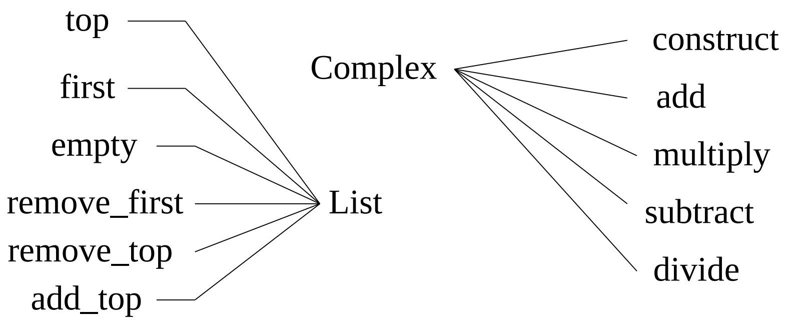

Figure 2.3: Type Based Object Identification

When we applied type based method on list.h, list.c and complex.c, we get Part Type relation Complex < List. The graph shown in Figure 2.3 is constructed for Figure 2.1. As we can see that there is no edge from *add_top* to structure Complex, because Complex is part type of List. It would identify as ADT1 = {{Complex}, {add, subtract, multiply, divide}} and ADT2 = {{List}, {top, first, empty, remove_top, remove_first, add_top}. Actually, global based and type based methods are used in combination, first global based method is applied on source code and then type based method is used for remaining functions.

2.3 Same Module Heuristic

This heuristic [5, 6] is based on programming conventions and is easy to apply. The heuristic groups together routines only with those data types in their signature and referenced global variables that are declared in the same module. In C, modules do not exist, but programmers simulate this concept by a header file test.h for the specification and a C file test.c for the body. Use of same module heuristic is dependent on how well programmers follow this convention.

This heuristic can be used to find ADT as well as ADO. When we apply this heuristic on our example program we get ADT1 = {{Complex}, {add, subtract, multiply, divide}}, ADT2 = {{List}, {top, first, empty, remove_top, remove_first, add_top} and ADO = {{stack,queue},{init, push, pop, enqueue, dequeue}}. This heuristic also combined stack and queue in one class. If stack and queue are implemented in different classes then it would identify as different classes.

2.4 Internal Access Heuristic

The purpose of an abstract data type is to hide implementation details of the internal data structure by providing access to it exclusively through a well-defined set of operations. This heuristic is based on this principle. Internal access [5] for type T means

- If T is an array, any index subscript is an internal access
- If T is a record, any field selection is an internal access
- If T is a pointer, any dereference is an internal access

Internal access looks at only internal accesses made by method, and does not consider formal parameters to associate method to type. The Internal Access heuristic associates types with those routines that have an internal access to them. In Figure 2.1 functions add, subtract, multiply, divide access internal variables of Complex and functions top, first, empty, remove_top, remove_first, add_top access internal variable of List. So, formed ADTs are {{Complex}, {add, subtract, multiply, divide}} and {{List}, {top, first, empty, remove_top, remove_first, add_top}. There are some problems with this method. For file datastructure.c in Figure 2.1, it did not detect the ADO for either stack or queue, because the routines do not access the internal record components of stack or queue. Another problem is method may access internal variables of two types. In this case it is difficult to decide to which class method should be associated. Other problem is that a method accessing internal data of type may just want to retrieve data to perform task which may not be related to type.

2.5 Scenarios for Identification of Objects

This process to identify object based on different scenarios is incremental. It takes part of the application and then identify objects using one of the following scenarios. These objects are implemented and code, which was previously used to implement the functionality provided by these objects, is discarded. During next iterations, other objects are identified and implemented in the same way. The three scenarios [17] those has been discussed in this method are called function driven objectification, data driven objectification and object driven objectification.

Function Driven In this method, functional part of the application is chosen and this part is converted into object oriented form. All other parts of the application, those use this functional part, are changed. All communications to this functional part are replaced by calls to methods in objects.

Data Driven In data driven objectification part, which operates on data, is identified and object is defined as data with functions working on that data. So, instead of functional parts, part selected by this method is data structures. These data structures can be variables, structures, files, databases, variable length records. Procedures which work on data structures become method of corresponding objects. If method works on more than one data structures then it has to be split in two or more methods. Sometimes it is difficult to split method, in this situation method may be associated to more objects and thus introducing redundancy or we can merge this two data structures into one. Like previous scenarios, after applying this method remaining procedural part of application needs to be changed to communicate with identified objects.

Object Driven Unlike previous scenarios, this scenarios is not driven by the existing code. It uses domain knowledge to identify objects. For some objects common sense dictates that they must be represented in particular system. The exact functionality associated with these objects has to be reverse engineered from the application code.

Problem with this method is that it requires more human interaction and is least suited for automation. Software engineer has to identify which part of code is functional. Object driven identification can't be done automatically because it requires domain knowledge.

2.6 Evidence Driven Object Identification

In this technique, first part is to identify candidate classes from source code. Once initial set of classes are identified, method attachment follows. This process [11] is incremental and iterative. In each step, objects are refined and classes are merged. This technique is called evidence driven because to attach method to particular class, it collects evidence for each alternative class using different type of analysis.

This techniques focus on two areas for identification of objects

- the analysis of global variable and their types.
- the analysis of complex data types in formal parameter lists.

Global Type Analysis In this analysis, global variables within modules are identified first. For decomposition of source code into modules any existing clustering techniques can be used. After identifying global variables in modules, their types are extracted using abstract syntax tree and candidate object classes are generated.

Data Type Analysis This analysis focuses on type definitions, like typedef C construct. Data types that are used in formal parameters of functions are also candidates for classes. The union of data types identified by global type analysis and data type analysis becomes initial set of classes.

2.6.1 Method Attachment

Once classes are identified, the next step is to associate methods to classes. For this, parameters of functions are analyzed. Basic types are ignored and only complex types are used for association. Rules for association are as follows

- For functions with no parameters, the return type and the type of global variables (in the scope of the particular function) that are modified/used become the initial candidate classes for which the particular function will become a method.
- For single parameter functions the parameter type along with the return type and the global types modified become the candidate classes for this method.
- For multiple parameter functions, all the parameter types along with the return type and the globally modified/used types are considered.

It is possible that function is associated to more than one classes. So, some method resolution techniques is required. The basic idea is to construct global evidence table per candidate class which have methods in conflict. To collect evidence for each method following analysis is done by these techniques.

State modification analysis In this analysis, if function modifies state of some object then method should be associated to that object. Because we would like methods that modify the state of their own class and minimize the side effects (state changes) of other classes.

Function call analysis Function call analysis focuses on the examination of data types in the actual parameter lists of function calls that occur within a body of the function that is to be considered as a method of a class. For example, if method M that corresponds to function F is in conflict and can be attached to different classes C1, C2, C3 generated from data types T1, T2, T3 respectively, function call analysis will examine the formal parameter lists of all function calls within the body of F. The data types that most often participate both on the formal parameter list of F and on actual parameter lists of calls within the body of F are considered primary candidate classes to attach method M.

Information flow For this first find the fan-in and fan-out of conflicting methods for each alternative designs. Fan-in means number of methods which call this conflicting method and fan-out means number of methods which are called from this method. Design which minimize the information flow is used to associate method to particular class.

Thus, this technique first finds candidate class using global and data type analysis and then attaches methods to particular class. For attaching methods to particular class, it collects evidence for that class by applying different analysis. Let us take sample code shown in Figure 2.1. Initial classes identified are {Complex, List}. Both have conflicting methods such as top, add_top, pop. Evidences are generated for both of the classes using different analysis and methods are associated to appropriate class. It has identified only two classes but by manual analysis there should be four classes. Problem with this method is it is not finding ADOs only finds ADTs.

2.7 Cluster Analysis to Identify Objects

The aim of this analysis [15] is to identify functionally related groups of record fields. To understand this technique, records with more number of fields are needed. So, consider system with four programs and one record containing nine fields. All fields(variables) are put in set of cluster items. For each variable, technique determines whether program is using it or not. Table 2.1 [15] shows matrix for this. If program is using some variable then entry corresponding to this is marked with 1 else 0.

To apply clustering on these variables, distance between variables is needed. To find distance between two variables, consider each row of Table 2.1 as one vector. So, we can find euclidean distance between any two variables. If we put distance between two variables in matrix, then we get distance(dissimilarity) matrix. Clustering algorithm takes this matrix as its input.

Agglomerative hierarchical clustering algorithm is used by this technique to cluster set of variables. This algorithm starts by putting each variable in own different cluster. Then two clusters, having minimum distance between them, are combined and this process continues. Finally only one cluster remains and algorithm terminates. This algorithm makes a tree. We start from leaves(each variable in separate cluster). At each stage of merging two cluster, one intermediate node is generated and at last root is generated. We can choose set of clusters as clusters which are created after N number of steps. N is chosen such that it result in best set of clusters.

There are two fundamental problems with clustering the record fields.

- When clustering, each item goes in exactly one cluster. However, sometimes one item is equally likely end up in more than one cluster. Consider two records which share only one key field. By applying this clustering technique, three clusters are created. One contains only key field, second contains fields from first record and third contains fields from second record. Perhaps we would have only two clusters as two records. Unfortunately, as items can go in exactly one cluster, this is not possible with this cluster analysis.

- When we build cluster hierarchy, sometimes there is more than one closest cluster. For example consider one cluster A which has two closest cluster B and C. This

	P1	P2	P3	P4
NAME	1	0	0	0
TITLE	1	0	0	0
INITIAL	1	0	0	0
PREFIX	1	0	0	0
NUMBER	0	0	0	1
NUMBER-EXT	0	0	0	1
ZIPCD	0	0	0	1
STREET	0	0	1	1
CITY	0	1	0	1

Table 2.1: The usage matrix that is used as input for the cluster analysis

algorithm chooses any of B and C arbitrarily. Suppose it chooses B at this point, but it is also possible that choosing of C at this point will give better result later.

Cluster analysis is used for object identification purpose. Each cluster represent candidate class. However, good classes are expected only with human interaction. At which step agglomerative clustering algorithm should stop is decided by engineer. Cluster analysis provides no information on which methods to attach to each of the classes identified.

Thus, cluster analysis can be used for restructuring record fields with some pitfalls.

2.8 Concept Analysis to Identify Objects

As with cluster analysis, concept analysis is also used for finding groups of record fields that are related in given application domain. Concept analysis also starts with table of indicating features and set of items. Then it partitions the set of items in set of disjoint clusters, depending on how many features they share.

Concept analysis [15] build concepts which are maximal group of items sharing certain features. It does not try to find single optimal grouping based on numeric distances, instead, it constructs all possible concepts.

To understand this concept, consider set A of items, set F of features and feature table $T \subset A \times F$ indicating features possessed by each item.

For set of items $I \subseteq A$ we can identify the common features

$$\sigma(\mathrm{I}) = \{\ f \in F \mid \forall i \in I : (i, f) \in T\}$$

For set of features $J \subseteq F$ we can identify the common items

$$\tau(\mathrm{J}) = \{\ i \in A \mid \forall f \in J : (i, f) \in T\}$$

top	({init, pop, push, enqueue, dequeue}, $\emptyset$)
c1	({init, enqueue, dequeue}, {Uses Queue})
c2	({init, push, pop}, {Uses stack})
bot	({init}, {Uses Queue, Uses stack})

Table 2.2: All concepts in example file datastructure.c

A concept is a pair (I, J) of items and features such that J = σ(I) and I = τ(J). In other words, concept is maximal collection of items sharing common features. Consider the example file datastructure.c of 2.1. Here items are functions and features are *Uses Queue* and *Uses Stack*. In our example, ({init, pop, push}, {Uses Stack}) is the concept of those items having feature Uses Stack. Complete set of concepts are given in Table 2.2.

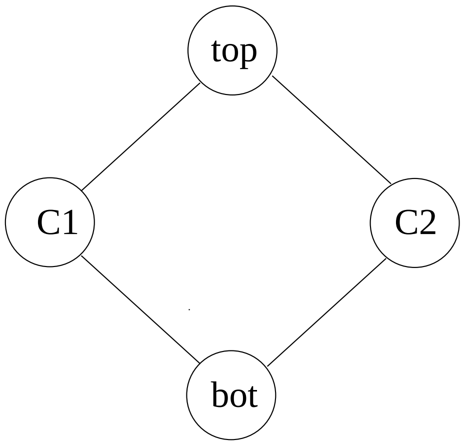

Figure 2.4: Concept Lattice

The concepts for given table forms partial order via

$$(I1,J1) \leq (I2, J2) \Leftrightarrow I1 \subseteq I2 \Leftrightarrow J2 \subseteq J1$$

This subconcept relationship allows us to organize all concepts in concept lattice, with meet $\wedge$ and join $\vee$ operation defined as

$$(I1, F1) \wedge (I2, F2) = (I1 \cap I2, \sigma(I1 \cap I2))$$
$$(I1, F1) \vee (I2, F2) = (\tau(F1 \cap F2), F1 \cap F2)$$

The visualization of concept lattice shows all concepts and subconcept relationship between them. Figure 2.4 shows the concept lattice for our example file datastructure.c. One possible partition of concept lattice is shown ({c1}, {c2}).

Chapter 3

Identifying Relationships Between Objects

Methods discussed in previous chapter try to capture compile time features i.e classes. When procedural code is converted into object oriented ones, object oriented features such as inheritance, association and polymorphism should be identified from procedural code. Maarit [9] has identified some methods to find object oriented features from procedural code. These methods identify how to find polymorphism from procedural code and also identify features emulating dynamic binding in procedural programs.

3.1 Identifying Class-Subclass Relationships using Variable length Record

The record or structure types of procedural program emulate the classes in object oriented programs. Similarly, unions or variable length records can be sources of class-subclass relationship. Consider the following union declaration.

```
union{
    byte mem_byte;
    int mem_word;
    char mem_array[10];
}mem_type;
```

The fields of the union are alternate to each other, so in object oriented program they become subclasses of same superclass. So, additional class is needed to be common superclass as shown in Figure 3.1.

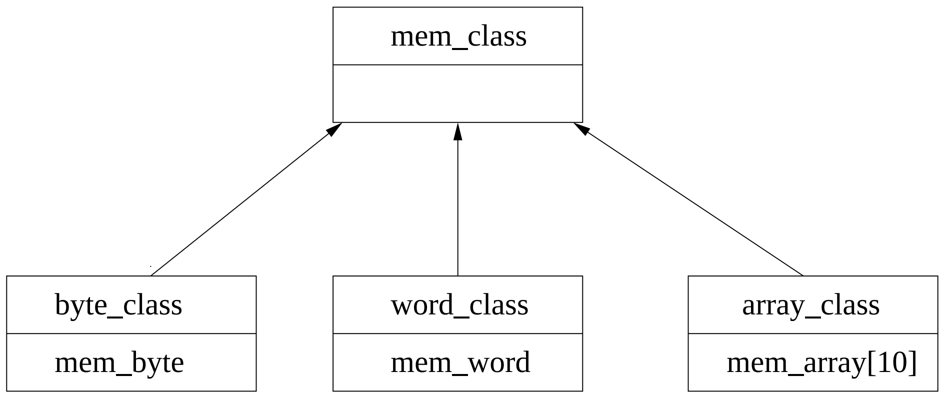

Figure 3.1: Classes generated from union definition

3.2 Identifying Dynamic Binding

In procedural code, the items emulating dynamic binding are the fields of variable-length records and the procedures manipulating these fields. However, the procedural code concerning the fields of the same record are typically in the same file, and procedures of the same file cannot have the same name in procedural code. Thus, the names of the procedures emulating dynamic binding are different, although their names in the produced object-oriented code should be the same. To understand this consider the example code given below.

```
switch(current_type){
        case MEM_BYTE: list_byte(mem_byte);
        case MEM_WORD: list_word(mem_word);
        case MEM_ARRAY: list_array(mem_array);
}
```

When type based object identification is applied on this code, operation *list_byte* belongs to *byte_class*, operation *list_word* to *word_class* etc. Maarit [9] has then applied condition based identification on this code to identify dynamic binding. Here, current_type is a global variable and takes its value from union field. It is also used in test of a case statement. In such case, names of all three operations has to be changed to have same name say list_mem. Also common superclass should have virtual function with same name. After doing this previous code is replaced by

```
super_mem.list_mem()
```

Here super_mem is instance of superclass mem_class and it points to byte_class, word_class or array_class depending on value in current_type. Thus appropriate function is selected based on actual value of super_mem.

3.3 Inheritance

Siff [14] has identified some interesting patterns those are used in C programs to simulate object oriented features such as inheritance, multiple inheritance. Procedural code such as C implements inheritance in various ways. This section discussed some of the ways to implement inheritance in C. In simplest way inheritance is implemented by redundant declarations. Consider following structure definition

```
typedef struct { int x,y; } Point;
typedef struct { int x,y,radius; } Circle;
```

Here Point is superclass of class Circle. These type of declarations are easy to identify. Another way of implementing inheritance is by using part type relation discussed in section 2.3. If A is used in declaration of type B then A becomes superclass of B. Consider the following function definition

```
typedef struct { Point p; int radius; } Circle;
```

In this case also Circle is subclass of class Point. Multiple Inheritance can also be implemented in same way.

3.4 Aggregation

Not all part type relations identify inheritance. If more than one instances of type A are used in declaration of type B then there can't be inheritance between A and B, but there should be aggregation relation between A and B. Consider following example

```
typedef struct { Point lt,rb; } Rectangle;
```

In this example Point can't be a superclass of class Rectangle, but here aggregation relationship exists between Rectangle and Point.

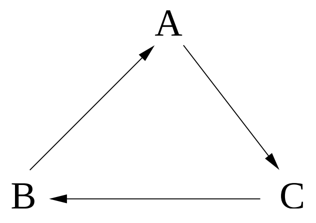
$$\begin{tikzcd} & A \arrow[dr] & \\ B \arrow[ur] & & C \arrow[ll] \end{tikzcd}$$

Figure 3.2: Dependency Graph for Sample Code in Figure 3.3

If dependency graph is drawn from part type relations and graph contains cycle then types enclosed in cycle can't form inheritance relationship between them.

Dependency graph G = (V, E) is directed graph and constructed from part type relation such that

$$V = \{S \mid S \text{ is a structure}\}$$
$$E = \{(S1, S2) \mid S2 \text{ is part type of } S1\}$$

Consider following example

```
struct B{
        A a;
        ...
}
struct A{
        C c;
        ...
}
struct C{
        B b;
        ...
}
```

Figure 3.3: Sample Code

Dependency graph for this code is shown in Figure 3.2. There is a cycle in dependency graph, so there should also be aggregation relationship between them. Harsha [13] has discussed some rules to distinguish inheritance and aggregation relationship in different cases.

Chapter 4

Proposed Method to Identify Abstract Data Objects

Global based and type based object identification are already discussed in section 2.1. Proposed approach just combines these two method and apply some refinement on these methods. This chapter discusses access graphs which are generated from source code. Once access graph is generated from source code, all subsequent analysis is done by using it. Method to identify ADOs using class cohesion is major refinement in existing technique.

4.1 Access Graph

Access graph G = (V,E) is generated for each function using function signature and function body. V is the set of vertex which contains all functions, all global variables and parameters and return type of all functions. E is the set of edges. There is an edge from node v1 to v2

- if v1 is a function and v2 is a global variable access by v1.
- if v1 is a function and v2 is a parameter of v1.
- if v1 and v2 both are functions and v1 calls v2.

There are six types of edge possible. Each edge in graph is marked with its type. They are

Read Edge If function is only reading the global variable or its parameter then edge is marked with read type.

Write Edge If function is only writing the global variable or its parameter then edge is marked with write type.

Read-Write Edge If function is reading and writing the global variable or its parameter then edge is marked with read-write type.

Return Edge This type of edge is between function and its return type.

Generate Edge If function's return type is different from all its parameters' type then edge between function and return type is called generate edge.

Call Edge If function v1 calls v2 then edge between v1 and v2 is called call edge.

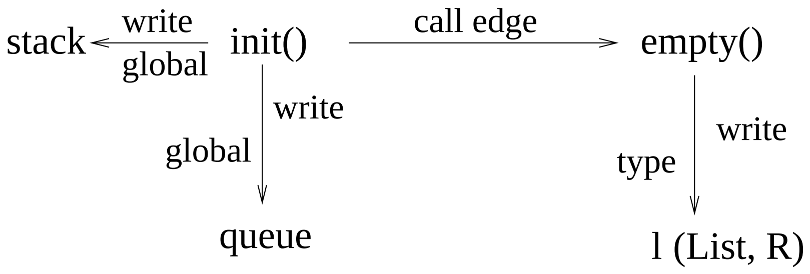

Figure 4.1: Access Graph

Edges also contain other informations. These informations can be extracted from the type of edge and using nodes connecting the edge. If edge is between function and global variable then it stores *global* tag and if edge is between function and its parameter or its return type then it stores *type* tag. All other edges contain null tag.

Nodes also contain some useful informations. Node corresponding to function parameter or return type contains type of the parameter and information that whether it is pass by value or pass by reference. If parameter is pass by value then node contains 'V' and if it is pass by reference then node contains 'R'. With all nodes some variable is also stored to indicate the type of the node, whether it is function, parameter, global variable or return type.

Figure 4.1 shows access graph for partial code given in Figure 4.2.

```
void empty(List *l){
l->len=0;
}
void init(){
empty(&stack);
empty(&queue);
}
```

Figure 4.2: Code for init and empty functions of Figure 2.1

4.2 Identify Abstract Data Object using Class Cohesion

Global based object identification, used to identify abstract data object, is already discussed in section 2.1, but object identified from this can be bulky. So, this section discussed some refinement applied to this technique.

First, candidate ADOs are found using the global based object identification. Access graph can be used for this, but only consider the edges which have *global* tag associate with them and nodes for function and global variable. Global variables, which are constant or initialized and not updated, are not consider in proposed technique. After finding final ADOs, these constant variables are added to class in which they are used by some function. Candidate classes identified using this technique can be bulky. This is because of initialization routine which access all global variables. Consider the sample code of Figure 2.1, init function initialize both stack and queue. So, only one class is created using global based identification but better solution is two classes, one for stack and one for queue. To divide this bulky class into more classes, concept of cohesion has been used.

Class cohesion [1] means *relatedness* of class methods. A highly cohesive class means class with one method. If class is found with low cohesion then it means class is bulky and it should be divided into more classes. So, class cohesion needs to be found after finding candidate ADOs. Following section discuss how to measure class cohesion.

4.2.1 Measuring Class Cohesion (TCC)

An Instance variable is directly used by method M if there is an edge from M to that variable in access graph. DU(M) is set of instance variable directly used by method M. An Instance variable is indirectly used by method M if there is an method M' such that M' is directly or indirectly uses that variable. In access graph if there is a path from method to variable then it is indirectly used by M. IU(M) is set of instance variable indirectly used by M. I(M) is the set of instance variable directly or indirectly used by method M.

$$\mathrm{I(M)} = \mathrm{DU(M)} \cup \mathrm{IU(M)}$$

Suppose there are n such sets I(1), I(2), ... I(n). If there exists one or more common instance variables between two methods then two methods are directly connected. Directly connected (DC) methods are found using

$$\mathrm{DC} = \{\ (\mathrm{I(i)}, \mathrm{I(j)})\ |\ \mathrm{I(i)} \cap \mathrm{I(j)} \neq \emptyset\ \}$$

When DC is to be find out, methods which access all instance variables are ignored from analysis because this method are mostly initialization function and share instance variables with all other methods.

Methods M1 and M2 are indirectly connected if there is a method M such that (M1,M) $\in$ DC, (M2,M) $\in$ DC and (M1,M2) $\notin$ DC. IC is the set of indirectly connected methods.

NP(C) be the total number of pairs of methods in class C. If there are N methods in class C then NP(C) is $N * (N - 1)/2$. NDC is the number of directly connected methods and NIC is number of indirectly connected methods.

$$\text{NDC(C)} = |\text{ DC(C) }| \text{ and NIC(C)} = |\text{ IC(C) }|$$

According to James *et al.* [1] Tight Class Cohesion (TCC) is relative number of directly connected methods.

$$\text{TCC(C)} = \text{NDC(C)}/\text{NP(C)}$$

Loose Class Cohesion (LCC) is relative number of directly and indirectly connected methods.

$$\text{LCC(C)} = (\text{NDC(C)} + \text{NIC(C)})/\text{NP(C)}$$

One another metric called **Lack Of Cohesion in Methods (LCOM)** [3] can also be used to find class cohesion. The definition of LCOM is as follows. Consider a class C with M1, M2 $\cdots$, Mn methods. Let I_i = set of Instance variables used by method M_i. There are n such sets I_1, I_2 $\cdots$, I_n.

Let P = $\{(I_i, I_j) \mid I_i \cap I_j = \emptyset\}$ and Q = $\{(I_i, I_j) \mid I_i \cap I_j \neq \emptyset\}$. If all n sets $I_1, \cdots,$ I_n are $\emptyset$ then let P = $\emptyset$. P contains pairs of methods which have no instance variables in common and Q contains pairs of methods which accessing one or more common instance variables.

$$\begin{aligned} \text{LCOM} &= |\text{P}| - |\text{Q}| \text{ if } |\text{P}| > |\text{Q}| \\ &= 0 \text{ otherwise} \end{aligned}$$

Larger the value of LCOM, greater the possibility that the class is less cohesive.

To split the bulky class into more classes this LCOM metric is used by harsha [13]. If LCOM is greater than 0 then class has to be split into more classes, but it is found that TCC is better than LCOM. Consider the class M which contains two methods M1, M2 and two member variables a and b. Access graph for this class is given in Figure 4.3.

If LCOM metric is found for this class then it is 1 - 0 = 1 and this says class is bulky and should be divided into more classes, but TCC for this class is 1/1 =1 and says class is cohesive and no need to split. It is easily seen from the access graph that class cohesive. Difference between these two metrics is that TCC considers the instance variables indirectly used by method M, while LCOM doesn't consider this. LCOM only takes into account instance variables directly used by method M, but instance variables indirectly used by method is also important to find class cohesion.

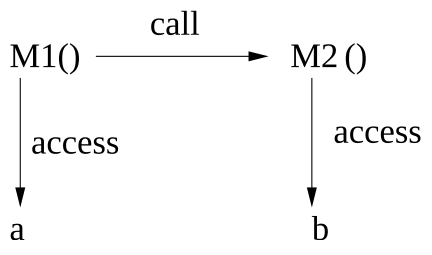

Figure 4.3: Access Graph to Compare LCOM and TCC metric

Only TCC is used in proposed method to find class cohesion. TCC of the ADO = {{stack,queue}, {init, push, pop, enqueue, dequeue}} identified for Figure 2.1 is 2/6 after ignoring the init function.

After finding ADOs, TCC of classes are measured. If TCC of any class is less than 0.5 then class is less cohesive and should be divided into more classes.
To divide the class C into more classes apply following algorithm:

1. Partition the member variables of class into two class A and B in all possible way.

2. Associate methods, which are only using variables of class A, to class A and methods, which only use the variables of class B, to class B. Methods which use variables from both classes are divided in two methods such that one uses variables of class A and second uses variables of class B.

3. Measure TCC of class A and class B. Also find the sum of these two TCC.

4. repeat step 2 and 3 for each possible partition found for class C in step 1.

5. If there is a partition of original class such that both new classes has TCC higher than 0.5 then choose partition with highest sum among them and stop the algorithm else choose the partition with highest sum and repeat this algorithm for class which has TCC less than 0.5.

When this algorithm is applied to example program of figure 2.1, only one partition is possible, one contains variable stack and second contains variable queue and init method is divided in two methods. Two classes A and B are created and both have TCC equal to 1. A = {{stack}, {init1, pop, push}} and B = {{queue}, {init2, enqueue, dequeue}}. Thus this algorithm divides the class with low cohesion into classes with high cohesion using the TCC metric.

Chapter 5

Proposed Method to Identify Abstract Data Types and Associate Functions to Classes

In this chapter, methods to extract Abstract Data Types (ADT) and associates functions to classes are discussed. The first step is to identify candidate ADTs. These candidate classes found do not have methods associate to them. So, methods to identify constructors and rules to associate remaining functions to candidate classes are discussed.

5.1 Identifying Candidate Abstract Data Types

All structures and type declarations in source code are candidate ADTs. For union or any variable length record, method described in section 3.1 can be used and all classes and subclasses become candidate ADTs. If type declaration is for a structure pointer or basic data type then that type declaration is removed from candidate classes. In case of structure, members of structure become attributes of ADT. Candidate classes are generated for char *, array of basic types and FILE *, if they are found in some functions' parameter's type, and parameters' types of all functions are found in access graph. These ADTs do not have methods associate with them. Let us take the sample C code of Figure 2.1, there are two structures Complex and List. Each of them is a candidate ADT. Consider the another example program string.c, which contains definitions of all functions which are in C header file string.h. This code doesn't contains any structure or type declaration, but one ADT, for char *, is identified from function parameter.

Next section discusses how to associate functions to classes. After associating functions to classes if there is a candidate class which doesn't contains any methods and also this class is not used anywhere in transformed code then remove such class. This is possible in case of char *, FILE * and structure which is already typedef. Consider the situation in which char * is passed as argument, so candidate class _char* is generated for it but if no methods are associated to _char* then there is no need of this class. If classes are remained for these type of ADTs then parameters of functions, which contains these types as argument, are changed. For example, if there is an class C which contains char * as its member and if parameter of some function is char * then it is replaced by C.

5.2 Rule Based Approach for Association of Functions to ADT

In this step, we need to associate functions to the ADTs, found in the previous step. Only the functions which do not access global variables are considered, as functions which access global variables are already covered while deriving ADOs. The approach used in assigning functions to ADTs is dependent on return type, arguments of functions and access to this arguments. Information, that the particular argument is only read by function or it is updated in function body, is also needed to associate function to ADT. All these informations are easily available from access graph which was discussed in section 4.1. If function has edge to both A and B and if A is part type of B then edge from function to A is not considered while associating function to classes. When we are associating function to ADT, it is possible that the function can become constructor in that ADT. So, first rules for identifying constructor are discussed and then rules for associating function to ADT are discussed. For each function, all rules are checked one by one. The first rule matched is applied to function. As ordering of the rules is important, consider that rules are assigned priorities and first rule has highest priority.

let F = $\{f_1, f_2 \cdots, f_n\}$ be set of functions which need to be associated to some ADTs found. ADT = $\{adt_1, adt_2 \cdots, adt_n\}$ are set of candidate ADTs found in previous section. Following are some definitions which are used to define function association rules. These definitions can be easily implemented as functions using access graph.

- NARGS(f_i) returns number of arguments of function f_i.
- ARG(f_i) returns set of ADTs in the argument list of function f_i.
- ARG(j, f_i) returns jth argument type of function f_i.
- RET(f_i) returns return type of function f_i, it returns $\emptyset$ if return type is not an ADT.
- PBR(i, adt) is true if adt $\in$ ARG(f_i) and adt is passed by reference to the function f_i.
- RA(i, j) is true if there exist a read edge from function f_i to ARG(j, f_i) in access graph and ARG(j, f_i) $\in$ ADT.

- WA(i, j) is true if there exist a write edge from function f_i to ARG(j, f_i) in access graph and ARG(j, f_i) $\in$ ADT.
- RWA(i, j) is true if there exist an read-write edge from function f_i to ARG(j, f_i) in access graph and ARG(j, f_i) $\in$ ADT.
- R(i, adt) is true if there exist at least one read edge but no write or read-write edge from function f_i to adt in access graph.
- W(i, adt) is true if there exist at least one write edge but no read or read-write edge from function f_i to adt in access graph.
- RW(i, adt) is true if there exist an read-write edge or read edge and write edge from function f_i to adt in access graph.
- GEN(f_i) is true if access graph contains generate edge from f_i to RET(f_i).
- Call(f_i) returns {functions which are called from f_i} $\cup$ {functions which f_i calls}.

5.2.1 Identifying Constructor

Some functions play the role of constructor in procedural code. Here some rules are proposed to find such functions. These rules also find class in which function can become constructor.

- **IF** (NARG(f_i) = 0) $\wedge$ (RET(f_i) $\in$ ADT) **THEN** f_i becomes constructor of RET(f_i).

  If function has no argument and return type is candidate ADT then this function becomes constructor in class generated from adt RET(f_i), because return type object is generated in this type of function. This is the most easy way to find constructor from procedural code.
  Consider the example

  ```
  struct Stack{
  int top;
  int element[100];
  }

  Stack empty(){
  ....
  }
  ```

  This function creates empty Stack and returns it. So, it becomes constructor in Stack class.

- **IF** (NARG(f_i) $\geq$ 1) $\wedge$ (GEN(f_i) = True) $\wedge$ (RET(f_i) $\in$ ADT) **THEN** it becomes constructor of RET(f_i).

  If all arguments of function are different from its return type and return type is a candidate ADT then this ADT is generated in this function using its arguments, so it becomes constructor in ADT created for return type.
  Consider following example

  ```
  Stack read_file(char *f){
          ....
  }
  ```

  Suppose, file f contains information to generate Stack and function reads file f, generate the Stack from it and returns Stack. Then, function becomes constructor in Stack class.

- **IF** (NARG(f_i) >= 1) $\wedge$ (R(i, RET(f_i)) = true) $\wedge$ (RET(f_i) $\in$ ADT) **THEN** it becomes constructor of RET(f_i).

  If arguments of function, whose type is same as return type (R), are only used in body and return type is a candidate ADT then function becomes constructor in this ADT. In this case, ADT (corresponding to return type) is generated using arguments of type R and other arguments. Read only condition of arguments of type R is necessary because if this condition is not hold then it is possible that some argument of type R is simply updated and returned, but in this case this function cannot be a constructor. Consider following example

  ```
  Complex add(Complex a, Complex b){
              ....
          a and b are only used
  }
  ```

  In this function a and b are only used not updated, So this function can become constructor which takes two complex numbers, add them and generate new complex number.
  Consider following function

  ```
  Stack pop(Stack s){
          s.top-=1;
          return s;
  }
  ```

This function removes the top element of the stack. It is not a constructor because it just updating its argument and returning it.

- **IF** (NARG(f_i) = 2) $\wedge$ (ARG(1, f_i) = ARG(2, f_i)) $\wedge$ ($\exists p, q$ | ((RA(i, p)) = True) $\wedge$ ((WA(i, q)) = True) $\wedge$ (PBR(i, ARG(q, f_i)))) **THEN** it becomes constructor of ARG(f_i) .

  If function has only two arguments and both are of same type (ADT) and if one of them is only used and second is only written then it is more chances that this function becomes constructor in that ADT. Here one argument is only written using other argument, so this is the constructor for that argument.
  Consider following example

  ```
  void infixtopostfix(char *infix, char *postfix){
          ....
  }
  ```

  In this function string infix is only used and string postfix is generated using infix string. So, this can be a constructor for ADT associated with char *. Same is true for function *char * strcpy(char * src, char* dest)* in string.c, here src is used and dest is generated using src.

5.2.2 Associating Functions to Classes

After applying rules given in previous section, if function is not identified as constructor then apply following rules to associate function to some candidate ADT.

- **IF** (NARG(f_i) $\geq$ 1) $\wedge$ (|ARG(f_i) $\cup$ RET(f_i) | = 1) **THEN** associate f_i to its argument type which is in ADT.

  If function has more than one argument but exactly one argument type is an ADT then simply associate function to its argument type without seeing that argument is updated or only used in that function. consider the following function

  ```
  int pop(Stack s){
  ....
  }
  ```

  This function returns top element of the stack. So, it is associate to class Stack.

- **IF** (NARG(f_i) >= 1) $\wedge$ (W(i, RET(f_i)) = True) $\wedge$ (RW(i, RET(f_i)) = True) $\wedge$ (RET(f_i) $\in$ ADT) **THEN** it associates to RET(f_i).

If function has at least one argument whose type is same as return type and function changes the state of that argument then function is more associated to that type than the other types which are also modified in function body.

- **IF** (NARG(f_i) $\geq$ 1) $\wedge$ ($\exists$j | (adt_j $\in$ ADT) $\wedge$ (PBR(i, adt_j) = True) $\wedge$ ((W(i, adt_j) = True) $\vee$ (RW(i, adt_j) = True)) $\wedge$ ($\forall$a $\in$ (ARG(f_i) $-$ adt_j) | (R(i,a) = True) $\vee$ (PBR(i,a) = False))) **THEN** f_i associates to adt.

  If function has arguments where only one type is being passed by reference and updated and all other types are only used in function or they are not passed by reference, then associate the function with the type which is updated.

- If function has more than one arguments which are passed by reference and also updated in function body then it is difficult to find the class to which function will be associated, because in this case function operates on more than one ADT. All ADTs, which are updated, are candidate class for that function. So, some resolution technique needs to be evolved. A metric to find class coupling is used here to associate method to one of the candidate class.

  Coupling Between Object class (CBO) CBO [3] for a class is a count of the number of other classes to which it is coupled. An object is coupled to another object if one of them acts on the other, i.e., methods of one use methods or instance variables of another. Since objects of the same class have the same properties, two classes are coupled when methods declared in one class use methods or instance variables defined by the other class.

  In order to improve modularity, coupling between the classes should be kept minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult.

  CBO for a class can be easily found using access graph, because for each function f, access graph already contains functions called from f, functions calling f and variables used by f.

  Apply following steps when function matches this rule:

  1. Let C be a set of candidate classes for function f_i.
  2. For each c$\in$C, associate f_i to c and measure the CBO_after(c) $-$ CBO_before(c), where CBO_before(c) denotes CBO of class c when f_i is not in class c and CBO_after(c) denotes CBO of class c when f_i is associated to c.
  3. Associate function f_i to class c which has lowest change in its CBO. If two or more classes have same change in their CBO then associate function to class which has lowest CBO_after.

Consider the example in which function F has two arguments A and B, and both arguments are ADT, passed by reference and also updated in F. Suppose function F is called by only functions F1 and F2 which are already associated to ADT B. If F is associated to A then CBO of B increases because in this case function of B calls the function of A and thus B is coupled with A. Now, if F is associated to B then there is no coupling between A and B. So, F is associated to B by using coupling metric.

- If none of the above rule matches then also apply coupling metric which is discussed in last rule and candidate classes are ADTs in function parameter and return type.
- If none of function parameter or return type is an ADT then again apply the coupling metric but here set of candidate classes are found in different ways. Let F be Call(f_i), then candidate classes are classes, to which functions from set F is associated.

5.3 Changing Function Declaration

There are two types of objects those are identified from procedural code, objects those are built around abstract data type (ADT) and objects those are built around abstract data object (ADO). When object is built around abstract data type, it contains that data type as its member. Function will be associated to ADT only if that data type is in formal parameter or it is in return type of function. After associating function to some ADT, function can directly access that data type because it is attribute of class. So, there is no need of ADT in function parameter or return type. If function is associated to ADO then function declaration doesn't change because ADO's attributes are global variables not the data types. So, function parameters cannot be extracted from object's attribute and can't be removed from its declaration.

Thus, when function associates to ADT, some parameter should be removed from function. This can be done automatically. We proposed the rules to change the function declaration when it is associated to ADT. This section discusses these rules with some examples, but these rules are not exhaustive some more rules are needed to consider all possibility.

5.3.1 Rule Based Approach to Change the Function Declaration

Following rules are applied when function f_i associates to some ADT.

- **IF** (NARG(f_i) $= 0$) $\wedge$ (RET(f_i) $\in$ ADT) **THEN** remove the RET(f_i).

  If function has no parameter and return type is user defined type, then function associates to its return type. Here, function is generating instance of its return type.

So, function becomes constructor in that class and constructor returns nothing. consider the following example

```
List empty(){
        List r;
        r.len=0;
        return r;
}
```

This is a function in procedural code. According to function association rule this function associates to class List and it becomes constructor in List. So, after association, function changes to

```
empty()
{
    len=0;
}
```

- **IF** (NARG(f_i) $\geq$ 1) $\wedge$ (RET(f_i) $\cap$ ARG(f_i) $\neq \emptyset$) $\wedge$ (PBR(i, RET(f_i)) = false) $\wedge$ (W(i, RET(f_i)) = True $\vee$ RW(i, RET(f_i)) = True) $\wedge$ ($f_i \Rightarrow$ RET(f_i)) **THEN** replace RET(f_i) by void.

  If function has at least one argument whose type same as its return type and those arguments are not passed by reference and also updated in function body, and function associates to its return type then replace return type by void. consider following example

```
List pop(List l){
        ....
}
```

  This function remove top element of l and return l. It is replaced in object oriented code by following function.

```
void pop(List l){
         .......
}
```

  We are not removing function parameter because it is passed by value. So, even if parameter is changed in function body, its effect is not reflected in outside code. If we remove parameter and call this function by this removed parameter then changes made by function body will be reflected outside the function and this pure function becomes function with side effects which contradict the functionality provided by this function in procedural code. So, return type is replaced by void not the function parameter doesn't change. Call *stack = pop(stack1)* is replaced by *stack.pop(stack1)*.

- **IF** (NARG(f_i) $\geq$ 1) $\wedge$ (GEN(f_i) = True) $\wedge$ (RET(f_i) $\in$ ADT) **THEN** remove return type.

  If return type is different from all function arguments and return type is part of ADT then function becomes constructor in class containing return type as its member.So, remove its return type.
  consider following example

```
struct node * read_node(FILE *fp){
        ....
}
```

  is replaced in object oriented code by

```
read_node(FILE *b){
        ....
}
```

  Call *struct node* head = read_node(fp)* is replaced by *node head(fp)*, where node is the class name containing struct node as its member.

- **IF** (NARG(f_i) $\geq$ 1) $\wedge$ ($\exists adt \in$ ADT, PBR(i, adt) = true $\wedge$ f_i => adt) **THEN** remove that adt.

  If there is a function parameter which is a pointer and function is associated to this parameter then remove this parameter from function declaration.
  consider following example

```
int strlen(char *b){
        ....
}
```

  is replaced by

```
int strlen(){
        ....
}
```

In this case, function parameter can be removed because it is pass by reference. So, changes made by function body to this parameter is reflected outside the function. So, we can directly call this function in object oriented environment by using that parameter as object. Call *i = strlen(s)* is replaced by *i = S.strlen()* , where S is a object containing **char* s** as member.

- **IF** (NARG(f_i) $\geq$ 1) $\wedge$ (RET(f_i) = empty) $\wedge$ ($\exists adt \in$ ARG(f_i) | $f_i \Rightarrow$ adt $\wedge$ (PBR(i, adt) = False) $\wedge$ (R(i, adt) = True)) **THEN** remove adt.

  If return type is void or some basic data type and function is associated to some parameter which is not updated in function body then we can remove that parameter from function declaration.

  consider following example

```
void test(List a, Queue b){
        ....
}
```

  In this example consider that function is associated to Queue then it is changed to following form in object oriented code.

```
void test(List a){
        ....
}
```

  And call *test(a,b)* is replaced by *b.test(a)*.

5.4 Consolidated Object Identification Technique

Following algorithm is proposed to convert procedural code into object oriented one.

1. Parse the source code and generate access graph as explained in section 4.1
2. Identify Abstract Data Objects
   (a) Apply global based identification technique with some refinements as suggested in section 4.2 to get ADOs
   (b) Measure Class cohesion (TCC) for each ADO as described in section 4.2.1
   (c) If TCC for some ADO is less then apply steps given in section 4.2
3. Identify candidate ADTs using structure, type definition and functions' parameters as given in section 5.1

4. For each function which doesn't access global variable

    (a) Apply the rules given in section 5.2.1 to see whether it can become constructor or not and if some rule matches then it becomes constructor of class specified by that rule

    (b) If function doesn't identify as constructor then apply the rules given in section 5.2.2 and associate function to ADT specified by first matched rule

    (c) Apply the rules given in section 5.2.3 to change the function declaration and change it as specified by first matched rule

5. Make the necessary changes in function body, like call other functions using object name etc.

Chapter 6

Conclusion and Future Work

A technique for identifying ADO and ADTs is proposed. Rule based approaches are also found to identify constructors in procedural code and associate functions to appropriate classes. Not all functions which can become constructors in object oriented code are identified by this technique; for this more analysis is needed. When functions are associated to some class, their declarations need to be changed; how this can be done is also discussed. So far, searching for constructors and changing function declarations have been nearly ignored in the literature. Techniques proposed in this report do these automatically.

Algorithms which combine all these proposed techniques are given and also applied on small examples. The results are close to manual analysis.

As future work, we can think of searching more object oriented features from procedural code and identifying patterns in procedural code to extract better design patterns. Currently, experiments were carried out on small examples and results were good. Experimentation is needed on large system to arrive at conclusion about the effectiveness of the methods proposed. We can think of applying these proposed techniques on parts of linux kernel and finally make the entire kernel object oriented. For this, entire linux kernel must be made g++ compatible, so that we can compile it after applying proposed techniques on some parts. Half of the kernel has already been made g++ compatible as previous work [16, 8].

We can easily see that all methods proposed in this report do not require human interaction and do all their tasks automatically. So, we can think of building a tool, which proposes candidate objects and relationship between them, indicates functions which are candidate for constructors, proposes association of other functions to classes and also indicates the changes in functions' declaration. However, final decision on what code should be bound with data must lie with the user having domain knowledge, to form appropriate objects. So, we can also think of providing facility in tool, which changes the object identification procedure based on user input. This means if user tells that some variables should be kept together in one class then tool considers this requirement first and based on this, applies identification procedure on procedural code.

References

[1] James M. Bieman and Byung-Kyoo Kang. Cohesion and reuse in an object-oriented system. In *SSR '95: Proceedings of the 1995 Symposium on Software reusability*, pages 259–262, New York, NY, USA, 1995. ACM Press.

[2] Gerardo Canfora, Aniello Cimitile, Malcolm Munro, and C. J. Taylor. Extracting abstract data types from c programs: A case study. In *ICSM '93: Proceedings of the Conference on Software Maintenance*, pages 200–209, Washington, DC, USA, 1993. IEEE Computer Society.

[3] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[4] Michael F. Dunn and John C. Knight. Automating the detection of reusable parts in existing software. In *ICSE '93: Proceedings of the 15th international conference on Software Engineering*, pages 381–390, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[5] Jean-Francois Girard and Rainer Koschke. A comparison of abstract data types and objects recovery techniques. *Sci. Comput. Program.*, 36(2-3):149–181, 2000.

[6] Jean-Francois Girard, Rainer Koschke, and Georg Schied. Comparison of abstract data type and abstract state encapsulation detection techniques for architectural understanding. In *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, page 66, Washington, DC, USA, 1997. IEEE Computer Society.

[7] Jean-Francois Girard, Rainer Koschke, and Georg Schied. A metric-based approach to detect abstract data types and state encapsulations. *Automated Software Engg.*, 6(4):357–386, 1999.

[8] Anil Gracias. Towards re-engineering of linux kernel from procedural to object-oriented. Master's thesis, Department of Computer Science, Indian Institute of Technology Bombay, 2002.

[9] Maarit Harsu. Identifying object-oriented features from procedural software. *Nordic J. of Computing*, 7(2):126–142, 2000.

[10] Ivar Jacobson and Fredrik Lindstrom. Reengineering of old systems to an object-oriented architecture. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 340–350, New York, NY, USA, 1991. ACM Press.

[11] Kostas Kontogiannis and Prashant Patil. Evidence driven object identification in procedural code. In *STEP '99: Proceedings of the Software Technology and Engineering Practice*, page 12, Washington, DC, USA, 1999. IEEE Computer Society.

[12] Sying Syang Liu and Norman Wilde. Identifying objects in a conventional procedural language: an example of data design recovery. In *ICSM '90: Proceedings of the IEEE International Conference on Software Maintenance*, pages 266–271, San Diego, CA, USA, 1990. IEEE Computer Society.

[13] Harsha Mahuli. Techniques for automatic identification of objects and their relationship from procedural code. Master's thesis, Department of Computer Science, Indian Institute of Technology Bombay, 2004.

[14] Michael Benjamin Siff. *Techniques For Software Renovation.* PhD thesis, University Of Wisconsin-Madison, 1998.

[15] Arie van Deursen and Tobias Kuipers. Identifying objects using cluster and concept analysis. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 246–255, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[16] Ashish Vanarse. Towards re-engineering of linux kernel from procedural to object-oriented. Master's thesis, Department of Computer Science, Indian Institute of Technology Bombay, 2005.

[17] Theo Wiggerts, Hans Bosma, and Erwin Fielt. Scenarios for the identification of objects in legacy systems. In *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, page 24, Washington, DC, USA, 1997. IEEE Computer Society.

Appendix

In this section, technique given in section 5.4 is applied to sample code and results are compared to results obtained by applying manual analysis on this code. Summary of code and some of the functions' declaration which are useful to explain results are given below.

```
functions from file www.cse.iitb.ac.in/~nadesai/mtp/string.c
some example functions' declaration

int strlen(char* str)
char* strcat(char* dest, char* src)
char* strcpy(char* dest, char* src)

struct Stack{
     int len;
     char* element[SIZE];
}

struct Queue{
    int front,rear;
    char* element[SIZE];
}

char* pop(Stack s);
void push(Stack s, char *str);
char* remove(Queue q);
void add(Queue q, char *str);

Stack queue_to_stack(Queue q);
Queue stack_to_queue(Stack s);

int main();
```

Figure 6.1: Sample Code for Experiment

If global based and type based object identification [12] is applied on this code then only one object is identified, which contains both structure Stack and Queue. This is because of functions queue_to_stack and stack_to_queue, which contains both structure Stack and Queue. Class for char* is not identified and functions accessing char* are not associated to any class.

If proposed technique is applied on this code then no ADOs are found because code doesn't contains any global variable. Two ADTs, Stack and Queue, are identified from structure definition and ADT for char* is identified from function parameter. When rules given in section 5.2 are applied on functions, result shown in Figure 6.2 is generated. Function queue_to_stack becomes constructor in Stack class as it matches second rule of constructor identification. If manual analysis is applied on code in Figure 6.1 then also this function becomes constructor in Stack class. Same is true for stack_to_queue. Function strlen becomes constructor because it generates dest string in its body. By technique also it is identified as constructor. All other functions are also associated to correct class by applying function association rules given in section 5.2.2.

By manual analysis same classes are generated with same members and methods. This technique is also applied on other small C programs and it is giving result close to manual analysis.

```
class C{
    char *s;
    int strlen()
    strcpy(C str)  // becomes constructor
    C strcpy(C src)
    .... other functions from string .c
}

class Stack{
    int len;
    C element[SIZE];
    queue_to_stack(Queue q) // becomes constructor
    C pop()
    void push(C str)
}

class Queue{
    int front,rear;
    C element[SIZE];
    stack_to_queue(Stack s)  // becomes constructor
    C remove();
    void add(C str);
}

int main();
```

Figure 6.2: Code after Applying Proposed Technique