2m11.2937.11

Université de Montréal

Objet Identification Using Conceptual Clustering

par

Shiqiang Shen

Département d'informatique et recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la faculté des études supérieures
en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en informatique

Novembre, 2001

© Shiqiang Shen

QA
76
U54
2002
v.003

Université de Montréal
Faculté des études supérieures

Ce mémoire de maîtrise intitulé

Object Identification Using Conceptual Clustering

présenté par

Shiqiang Shen

a été évalué par un jury composé des personnes suivantes:

Président: Balázs Kégl
Directeur de recherche: Houari Sahraoui
Membre: Petko Valtchev

Mémoire accepté le: 15 JANVIER 2002

Acknowledgements

I want to thank my supervisor, professor Houari Sahraoui, for his support over my whole thesis period. I really appreciate the chance to work in such an interesting project under his valuable guidance and advice. He has constantly supported me in very kind and encouraging way by pointing out relevant research, generating interesting ideas. I am thankful to Idrissa Konkobo, Salah Bouktif for many interesting discussions on the topics of my thesis that have broadened my perspective.

Abstract

The object identification is one crucial step in migrating legacy systems to the object oriented technology. It aims at reducing the overwhelming number of lines of code in the legacy systems to a number of high-level design decision, and tends, thus to minimize the need for domain application experts. This thesis proposes a novel approach for identifying objects in an automatic fashion. It differs from other work by the fact that it borrows part of its inspiration from the artificial intelligence sub-field of conceptual clustering and focuses on the identification of objects which should be internally cohesive meanwhile inter-dependency between them should be kept as loose as possible.

We identify two kinds of structural information to be used by the approach, one is the usage of each variable in a set of routines, the other is the call relation among routines. The two relations are supposed to significantly capture the basic structural information leveraged for object identification.

The prototype we built can work in an automatic fashion. It is also open to human intervention when an expert is available. Those adjustable parameters available in the prototype offer more flexibility in searching for candidate objects similar to those already found.

We introduce a process of iterative optimization into the approach to seek higher quality of objects decomposed from a legacy system. The process compares the qualities of two sequential clustering trees according to their total sum of couplings between candidate objects and terminates when there is no further improvement in clustering quality.

The approach adopts the new clustering criterion functions (objective functions) that are directly associated to the well-known object oriented design metrics (cohesion

and coupling), introduces the multi-criteria decision-making to insure all identified objects with a relatively high degree of cohesion and then to keep its coupling with others as low as possible.

In order to overcome order effects, we put a *Reordering* procedure into the iterative optimization process and extract the so-called interleaved ordering in each iteration. The introduction of reordering procedure to our approach mitigates ordering effects globally and uncovers better clustering effectively.

The approach has been applied to the three small real-life systems and the results are compared to those identified manually. The validation results of the approach are considered to be good concerning recall rate. The further analysis on the results reveals its drawback due to the inherent limitation of automated techniques.

Keywords: conceptual clustering, re-engineering, object identification, legacy software migration.

Sommaire

L'identification des objets est une étape cruciale dans la migration des systèmes légués vers les technologies orientées objet. Elle vise à réduire le nombre de lignes de codes très élevée dans les programmes légués à un nombre de décisions de conception de haut niveau; Elle tend donc à minimiser le besoin d'experts du domaine. Cette thèse propose une nouvelle approche pour l'identification des objets de manière automatique. Elle diffère des autres travaux par le fait qu'elle emprunte une part de son inspiration à un sous domaine de l'intelligence artificielle: le regroupement conceptuel; elle se concentre sur l'identification des objets qui doivent être cohésifs tout en conservant une interpédance qui soit la plus basse possible.

Nous identifions deux sortes d'informations structurelles à utiliser par cette approche, l'une est l'utilisation de chaque variable dans un ensemble de routines, l'autre est représentée par les relations d'appel entre les routines. Les deux relations sont supposées capturer les informations structurelles de base pour l'identification des objets.

Le prototype que avons construit peut fonctionner de manière automatique. Il est aussi ouvert à l'intervention humaine lorsqu'un expert est disponible. Ces paramètres ajustables offrent plus de flexibilité dans la recherche d'objets candidats similaires à ceux déjà trouvés.

Nous introduisons un processus d'optimisation itérative dans l'approche dans le but de trouver des objets de meilleurs qualités dans un code légué. Le processus compare les qualités de deux arbres de regroupements séquentiels en fonction de la somme totale des couplages entre les objets candidats et se termine quand il n'y a plus d'améliorations qualitatives possibles dans les regroupements.

L'approche adopte un nouveau critère de regroupement (les fonctions objectifs) qui sont directement associées à des métriques orientées objets bien connues (la cohésion

et le couplage) et introduit la notion de prise de décision multi objectifs dans le but d'assurer que chaque objet identifié a relativement un haut degré de cohésion et conserve un couplage aussi bas que possible, par rapport aux autres objets.

Dans le but de résoudre le problème de l'effet de l'ordre, nous avons introduit dans le processus d'optimisation itérative, une procédure qui réordonne et nous avons extrait les ordres dits "**interleaved**" dans chaque itération. L'introduction de cette procédure à réduit les effets de l'ordre et permis d'avoir de meilleurs groupes.

L'approche a été appliquée à trois systèmes de petites tailles et les résultats ont été comparés à ceux identifiés manuellement. D'un point de vue "**du taux de rappel**" les résultats de validation sont bons. Une analyse approfondie des résultats révèle que les inconvénients sont dus aux limitations inhérentes aux techniques automatiques.

Mots clés: Regroupement conceptuel, retro ingénierie, identification des objets, migration des applications léguées.

CONTENTS

Acknowledgements i

Abstract ii

Chapter 1 Introduction 1

1.1 Motivation 1

1.2 Goals 2

1.3 Outline 3

Chapter 2 State of the Art 5

2.1 Terminology of Reengineering 5

2.2 The Importance of Object Identification in Re-engineering 7

2.3 Overview of Current Techniques of Object Identification 8

2.3.1 Global-based Object Identification 8

2.3.2 Type-based Object Identification 12

2.3.3 Unifying Forward and Reverse Engineering 13

2.3.4 Other Methods to Identify Objects 16

2.4 Problems in Object Identification 19

Chapter 3 Conceptual Clustering 21

3.1 A Brief Introduction to Conceptual Clustering 21

3.2 Cobweb System 23

3.3 An Example of Cobweb 26

3.4 Some Important Properties of Cobweb system 27

Chapter 4 Conceptual Clustering Based Approach 29

4.1 Overview of the Proposed Approach 29

4.2 An Example of Applying Cobweb to Object Identification 31

4.3 Limitation of Applying Cobweb to Object Identification 33
4.3.1 Cobweb Algorithm Itself 33
4.3.2 Object Function CU 33
4.4 Conceptual Clustering Based Approach 34
4.4.1 Approach Overview 34
4.4.2 Two Relations 36
4.4.3 Clustering Criterion Functions of OI-Cobweb 38
4.4.4 Multi-Criteria Decision Making 40
4.4.5 OI-Cobweb Algorithm 43
4.4.6 Reordering Procedure 44
4.4.7 An Example of the Approach 45

Chapter 5 Implementation 56
5.1 Supporting Tool 56
5.2 Extracting Information 59
5.3 Properties of the Prototype 64

Chapter 6 Evaluation of the Algorithm 69
6.1 Systems Studied 69
6.2 Reference Corpus 71
6.3 Classification of Matches 72
6.4 Accuracy and Recall Rate 75
6.5 Case Studies 78
6.5.1 Case Study 1 78
6.5.2 Case Study 2 79
6.5.3 Case Study 3 79
6.5.4 Summary of Evaluation Results 80
6.6 Discussions and Lessons Learned 83

Chapter 7 Conclusion 86

7.1 Thesis Summary 86
7.2 Future Work 88

Bibliography 90

LIST OF FIGURES

Figure 1 Reference relation between routines and global data 9
Figure 2 Routine interdependence graph of the relation of figure 1 9
Figure 3 Reference graph of the relation between routines and global data 10
Figure 4 A Cobweb clustering for four one-celled organisms 27
Figure 5 Overview of the object identification approach 30
Figure 6 Clustering tree for object identification 32
Figure 7 An Imaginary System 37
Figure 8 VR-relation from An Imaginary System 37
Figure 9 RR-relation from An Imaginary System 38
Figure 10 A Scenario of Three Clusters 40
Figure 11 Procedural Code of Collections in C 47
Figure 12 Matrix representation of a set of relations between routines and variable 48
Figure 13 Matrix representation of a set of relations between routines 49
Figure 14 Initial Clustering of Variables in A Random Order 52
Figure 15 Clustering of Variables in A New Order in the Second Iteration 53
Figure 16 Clustering of Variables in A New Order in the Third Iteration 54
Figure 17 Different Clustering in different presented ordering 55
Figure 18 An Example in Brower of Discover's DeveoperXpress 57
Figure 19 An Example in View of Discover's DeveoperXpress 58
Figure 20 A sample of Input File for the Approach 64
Figure 21 Interface of the prototype 65
Figure 22 Parameter Setting 66
Figure 23 Suite of Analyzed C Systems 70
Figure 24 Summary of model elements for reference corpus 72
Figure 25 Example correspondences of candidates and references 75
Figure 26 Summary of evaluation results 80
Figure 27 Evaluation results in percentage 81
Figure 28 Overall quality of the approach 82

Figure 29 Summary of CCBA and GOAL evaluation results 82
Figure 30 Sensitivity of the approach to COHESION variations 83

Chapter 1 *Introduction*

1.1 Motivation

The software industry has a lot of legacy systems that have been used and maintained for several (or even dozens of) years. After undergoing a number of changes over the years, in most cases, they tend to become ill-structured, highly redundant, poorly self-documented, and weakly modular. These systems are important and valuable from the business point of view and still remain in heavy use. However, mere maintenance of these legacy systems is not always adequate and even not feasible, instead they require larger modernization and re-engineering [37].

In order to address these issues, many organizations have been migrating their legacy systems to object-oriented technology. This migration goal has strategic relevance because the demand for maintenance/evolution of existing systems is likely to increase rapidly in the near future as a result of major innovations in both the administrative processes and the technologies in use. Advantages of object-oriented programs are considered to be encapsulation, data abstraction, information hiding, etc. In addition, they are easy to understand, maintain, and reuse. Re-engineering

these systems to create object-oriented architectures seems to be a more feasible approach [54].

More than 50% of the time needed for the re-engineering of an existing legacy system is spent in understanding the program before the actual change can be designed and realized, as several case studies have shown [17]. When re-engineering non-object-oriented (procedural) programs into object-oriented ones, it is important to identify objects or components from procedural programs in reverse engineering activities. The identification of them can help in comprehending the system's design and, in particular, the structure and meaning of the data. When the cost of manual identification of objects is prohibitive, the automated identification of objects is more economical especially for large systems, and it allows engineers to skip over implementation details that hide, or make difficult to recognize, the meaning of the system's parts.

Research in the area of object identification has yielded many techniques, and their number is still growing. However, a simple automatic solution to object identification seems to be few and the quality of the automatically identifying objects is not good enough to be accepted without additional examination by the maintenance engineers [36].

1.2 Goals

Aiming at achieving better results, we propose a new and more flexible approach, namely *conceptual clustering based approach*, to identify objects automatically from legacy systems. In this thesis, we especially pay attention to those aspects that are supposed to be primary factors affecting or improving the quality of automated techniques for this problem. The principal new results of this thesis include:

- Maximal use of useful information extracted directly from legacy code which is supposed to significantly capture the basic structural information leveraged for object identification;
- Significant practical experience with the use of conceptual clustering for object identification;
- Association of clustering criterion functions of object identification to well-known design cohesion and coupling metrics for object-oriented systems that is incorporated into object identification;
- Introduction of multiple criteria decision making into the process of optimizing data grouping which makes the approach more flexible and enhances the quality of object identification;
- A discussion of a number of problems (and solutions) involving use of the approach in the case studies;
- Validation of the approach based on comparison of the results obtained by the approach with those identified by human from three small real-life systems;
- Implementation of a prototype tool that uses the approach to find potential objects and also provides the user with the other base facilities that may support more typical object identification tasks through setting parameters based on the characteristics of the subject system.

1.3 Outline

The rest of this thesis is organized as follows:

Chapter 2 introduces the terminology and concepts of reengineering used in the thesis and presents existing techniques for object identification.

Chapter 3 first gives a brief introduction to conceptual clustering, and then describes Cobweb system, a conceptual clustering technique, which our proposed object identification approach heavily relies on.

Chapter 4 describes the conceptual clustering based approach we propose, discusses those concerns about the approach's efficiency, and shows how this approach works by applying it to a well-known example in this area.

Chapter 5 presents the prototype of the approach, and describes some aspects of base facilities the prototype provides.

Chapter 6 performs evaluation of this approach with three small real-life systems, compares the results obtained by the approach to those identified manually in order to assess its accuracy, strength, and effectiveness.

Chapter 7 makes some conclusions from considering the benefits gained by using the approach in the thesis, and then discusses future work.

Chapter 2 State of the Art

This chapter considers re-engineering in general and object identification in particular. We first introduce the terminology. Then we explain the importance of object identification in re-engineering, and after that we present published approaches of object identification and discuss the problems of these approaches.

2.1 Terminology of Reengineering

In this section, we introduce some basic concepts concerning reengineering, which are based on [10]. For simplicity, the definitions assume that the software life cycle consists of three phases: requirements analysis, design, and implementation.

Software maintenance

The ANSI definition of software maintenance (ANSI/IEEE Std 729-1983) is the "modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment". The first step in software maintenance is to examine the program to understand it. Reverse engineering facilities can be used to support the maintenance process. Thus, reverse engineering is the part of the maintenance process helping to understand the

program in order to make the desired changes. Maintenance can also be considered as reuse-oriented software development [1].

Forward Engineering

Forward engineering is the traditional process of moving from the requirements of the system to its design, and from design to the concrete implementation of the system. Actually, forward engineering means exactly the same as engineering. The adjective forward is used to distinguish the term from reverse engineering.

Reverse engineering

Reverse engineering is a reverse process for forward engineering. In reverse engineering, the extracted information about a system is more abstract than the system itself under examination. For example, abstractions or design decisions are generated from the implementation level. Reverse engineering can start from any level of abstraction or at any stage of the life cycle. Reverse engineering does not involve changing the subject system. It is a process of examination, not of change or of replication.

Restructuring

Restructuring is the transformation from one representation form to another at the same abstraction level. The transformation preserves the external behavior of the system. Restructuring is typically used in implementation stage to transform code from an unstructured form to a structured form (transforming, for example, goto-statements to control structures). In addition, restructuring can be used in other stages, for example to reshape design plans or requirement structures.

Design recovery

Design recovery is a subset of reverse engineering in which domain knowledge, external information, and deduction or fuzzy reasoning are added to the observations

of the subject system to identify meaningful higher level abstraction beyond those obtained directly by examining the system itself.

2.2 The Importance of Object Identification in Re-engineering

Analyzing old software has become an important topic in software technology, as there are millions of lines of codes which lack proper documentation; due to ongoing modifications, software entropy has increased steadily. If nothing is done, such software will die of old age — and the knowledge embodied in the software is inevitably lost.

As a first step in legacy system, one must reconstruct abstract concepts from the source code. In a second step, one might try to transform the source code such that the structure of the system is improved and obeys modern software engineering principles. One particular problem is modularization of old code. Old systems have not been developed by today's modularization criteria. Therefore, static information like control and data flow, access to non-local objects, or interface information must be extracted in order to guide restructuring. Modularization can then be achieved by manual changes or automated program transformation or both [31].

Although procedural programming languages do not directly support object-oriented programming constructs, they do provide several object-like features (such groupings or related data, abstract data types, and inheritance). It would be very useful to the maintenance programmer to understand the data and function relationships and objects the original designer had in mind. Clustering and re-engineering operations on the components belonging to a candidate object are necessary to transform them into an actually reusable object. Furthermore, success maintenance requires a precise knowledge of the data items in the system, the ways these items are created and modified, and their relationships. As Livardas and Johnson pointed out [40], identifying objects in procedural code helps to

- understand system design;
- test and debug;
- re-engineer the system from a procedural program into an object-oriented one;
- avoid degradation of original designs during maintenance;
- facilitate the reuse of existing methods contained in the system.

There is a lot of work concerning the migration from procedural languages into object oriented ones. Actually, most of the papers concerning this topic use the term object identification. To be exact, objects can be considered as run-time features and classes as compile-time features. We usually search for object-oriented features from code. Consequently, we can find compile-time features (classes). However, in this thesis we use both the terms class identification and object identification to mean very much the same.

2.3 Overview of Current Techniques of Object Identification

This section describes different approaches for identifying objects in procedural programs.

2.3.1 Global-based Object Identification

Liu et al. have used global-based object identification in their system called Object Finder [39, 47]. The method acts as follows. For each global variable of the program, the set of routines that directly use the global variable is determined. After that a routine interdependence graph is constructed, which show the dependence between routines consequent to their common coupling to the same global variable. A node $P(x)$ in the graph denotes the set of routines that reference a global variable x. An edge between $P(x_1)$ and $P(x_2)$ means that the two sets are not disjoint ($P(x_1) \cap P(x_2) \neq \emptyset$). Figure 1-a shows the reference relations between the routines r_i's and the

global data d_i's of a program. Figure2 shows the corresponding routine interdependence graph. Such a kind of graphs is used to identify objects. Each isolated sub-graph is considered to embody an object.

	d_1	d_2	d_3	d_4	d_5	d_6
r_1			1			
r_2	1		1			
r_3		1				1
r_4		1				
r_5	1					1
r_6					1	
r_7	1					
r_8		1		1		
r_9				1		
r_{10}					1	
r_{11}	1		1			

Figure 1 Reference relation between routines and global data

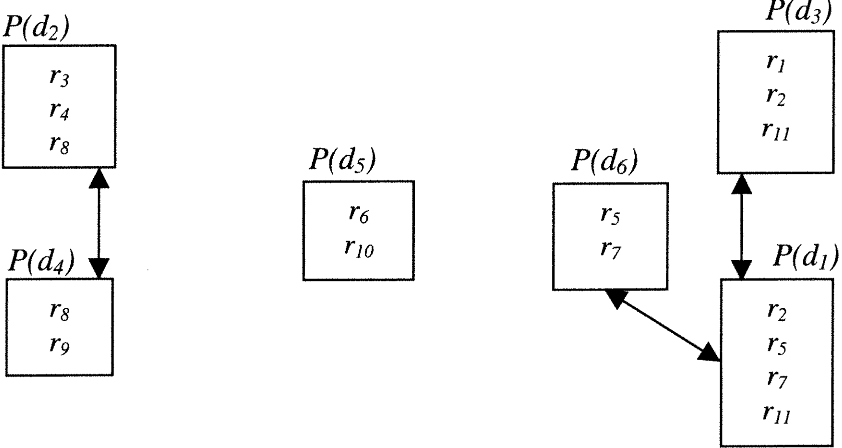


Figure 2 Routine interdependence graph of the relation of figure 1

Dunn and Knight introduce a different way to construct the graph consisting of global variables and routines [15]. In reference graphs, nodes are either routines or global variables, and an edge between a routine and a variable means that the routine uses the variable. Figure 3 shows the reference graph of the relation of figure 1. Like the routine interdependence graph, this kind of graph is used to identify objects through isolated sub-graphs.

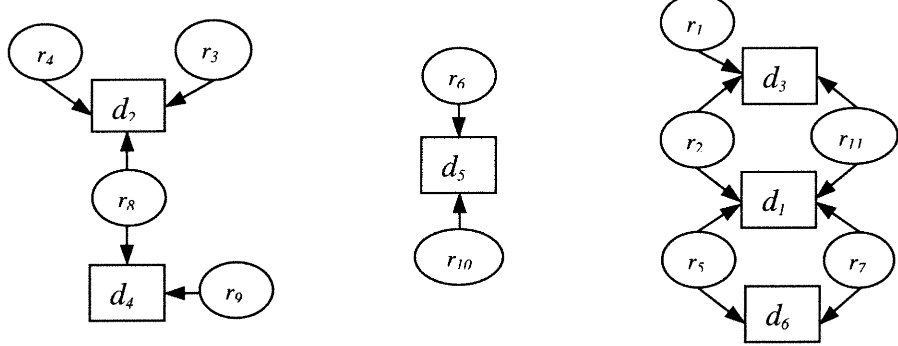


Figure 3 Reference graph of the relation between routines and global data

When applying global-based identification methods, the program code may be so tightly coupled that the whole program is one large connected component. Consequently, the method cannot split the program into smaller parts. In these situations, the user can specify (by hand) those global variables that form undesired links between nodes (routines) in the graph. These global variables are then ignored during the global-based object identification analysis.

Many other object identification systems have applied and developed global-based identification further. For example, Livadas and Johnson have detected some deficiencies from the method [40]. First, in programming languages allowing nested procedures (like Pascal), those variables that are visible for several procedures should be considered as global variables. Second, although a global variable is

passed as a parameter to a function, the receiver function should notice that the parameter originally is a global variable.

In the extended method proposed by Livadas and Johnson [40], the internal program representation is a system dependence graph. It is a directed labeled multi-graph consisting of a program dependence graph and a collection of procedure dependence graphs. The program dependence graph represents the main program, and procedure dependence graphs represent procedures. Each node of these graphs represents a program construct such as declaration, assignment, etc.. Additional nodes represent, for example, formal and actual parameters. The edges represent several kinds of dependences among the nodes and are distinguishable according to the labels attached to them. When using the system dependence graph, all the information needed can be obtained from the headings of the procedures. Thus, procedures with lacking bodies (for example library procedures) can also be modeled with the system dependence graph.

Canfora et al. use global-based object identification in their RE^2 project [5, 6, 7, 8]. The authors propose an algorithm that transforms a reference graph into a set of strongly connected and disjoint sub-graphs, where each sub-graph represents a candidate object. This transformation is based on the notion of variation of the internal connectivity of sub-graphs (ΔIC) of the reference graph. At the beginning, each routine defines a sub-graph (with a ΔIC). Two primitives are used to transform the graph (in an iterative way): *Merge* and *Slic*e. *Merge* clusters all the data of a sub-graph into a single data node. This is done when the ΔIC is greater than a threshold value. *Slic*e, consists of slicing a routine to dissociate two sub-graphs [32, 55, 56]. This occurs when the ΔIC is less than the same threshold value. The major weakness of this algorithm is the way the threshold value is calculated. The proposed approach is based on the programming style, which is very difficult to assess.

2.3.2 Type-based Object Identification

Liu et al. also provide another method to identify objects in Object Finder [39, 47]. This method is based on types and acts as follows. First, a topological ordering of all types in the program is defined as follows. If a type **x** is used to define a type **y**, then we say that **x** is a part of **y**, and **y** contains **x**. The relation of types is transitive: if **x** is a part of **y** and **y** is a part of **z**, them **x** is a part of **z**. Second, routines and types are connected together. A type is connected to a routine, if it is a formal parameter or a return type of the routine. However, if a type and its part type would be connected to the same routine, the part type will be ignored. The groups consisting of connected routines and types form objects and their routines. Again, human assistance is needed to split too large objects.

The reason why part types are rejected as described is to eliminate some irrelevant connections. We may, for example, have a type **node**, and another type **stack** whose items are nodes. Thus, **node** is a part of **stack**. Consider a routine **push** having two arguments: one of type **node** (which will be pushed) and another of type **stack** (to which a node will be pushed). For a human, it is clear that **push** should be a routine of **stack**, not a routine of **node**. If we follow the above rules, the connection between **push** and **node** will be removed, and the remaining connection will be the one between **push** and **stack**, as we intended.

Livadas and Johnson have noticed some shortcomings in type-based object identification, too [40]. The return type of a procedure is not always appropriate criterion for forming connections. For example, a routine **is_empty** (for a **stack**) has a Boolean return type. However, the routine is not a routine of Boolean, but a routine of **stack**.

Livadas and Johnson also introduce the method of receiver-based object identification [40]. Receiver parameter type of a routine means a type whose parameter is modified in at least one execution path of the routine. If a routine has several parameters, one of which is modified, the type of that parameter is typically an object candidate. For example, consider a routine **`push`** having two arguments: one type **`node`** and another of type **`stack`**. The routine pushes the given **`node`** to the given **`stack`**. Thus, only the parameter of type **`stack`** is modified, and **`stack`** should be an object candidate having the routine **`push`**.

Other researchers, such as Yel et al. (1995), Girard et al., (1997) proposed some heuristics to enhance Liu and Wilde's work. Yel et al. combine data structures with global variables in order to form groups of routines, data structures and global variables. Girard et al. propose an approach called similarity clustering to identify both objects and classes. This approach is based on the similarity metric introduced by Schwanke (1991).

2.3.3 Unifying Forward and Reverse Engineering

Gall et al. introduce CORET or COREM (Capsule Oriented Reverse Engineering Technique/Method) to translate non-object-oriented programs into object-oriented ones [18, 19, 20, 21, 22, 23, 24, 25, 26, 34]. CORET not only considers data stores as object candidates, but also examines functional relationships between data structures to get more object candidates from which several are selected to become objects. CORET divides object recovery into four phases: design recovery, application modeling, object mapping and source-code adaptation.

The first phase, design recovery generates different low-level design documents (i.e. structure charts, data flow diagrams). These documents are again modified to generate an entity-relationship diagram (ERD). The ERD consists of entities based

on the data stores of the data flow diagram. Data structures that are functionally related to these data stores are also added as entities of the ERD. A functional relationship is defined as a manipulation or use of one or several attributes of an entity (e.g. data store) by another entity (e.g. data structure). The relations of the diagram are both special relations (is-a, part-of) and general relations. The general relations are derived from those procedures of the program that incorporate a functional relation between two entities.

The entity-relationship diagram is transformed into an (reversely generated) object-oriented application model. The transformation is based on the structural similarities of these two design representations: each entity is mapped to an application object. The relationships (is-a, part-of) between objects and operations of objects are derived directly from the corresponding structures of the entity-relationship diagram. They are also derived from the declaration parts of the source code: is-a relations can be derived from variant records while part-of relations can be derived from array or pointer structures within data type definitions. The operations for an object can be found both from entity-relationship diagram and by the examination of the source code to recognize the procedures manipulating the object.

The second phase, application modeling constructs an (forwardly generated) object-oriented application model. This model is independent of the actual procedural implementation, but is based on the same requirements as the examined methods [11, 12, 49]. Actually, this step models the application, which is earlier modeled by the means of procedural program design, again by using object-oriented techniques.

The third phase, object mapping maps the objects of the two models (reverse and forward) together. The goal of this step is to find a mapping of similarities between the elements of those two models. The papers [25], [26] especially concentrate on finding the similarities using different binding techniques. Because of the different

origins, the models have some differences. The forward model originates from the requirements analysis and the domain knowledge, and therefore does not have detailed information (e.g. attributes have no types). The reverse model on the other hand offers a lot of detailed information because it originates from the examined source code. Thus, in mapping objects, the forward model is working as a pattern for the target application model to which the reverse model gives the detailed information. The two models have varying amount of attributes, instance connections, and message connections. Thus, some parts remain without correspondence. These parts contain elements of the procedural program that cannot be mapped to the target object-oriented model, and they form so-called procedural remainder.

In the fourth phase, source-code adaptation, both the earlier mapped elements and the procedural remainder are adapted to object-oriented concepts. Syntactically, objects are formed by encapsulating the attributes and the declaration parts of the procedures. Global data items are encapsulated in separate objects, too. They are called data objects, because they have only attributes, but no procedures. The procedural remainder produces the following objects. First, it gives master objects which include the highest level of system control and can be compared to a control unit. A typical example is the main program. Second, it gives aggregation and coordination objects, which perform system control over a specified set of procedures and provide some kind of functionality for a master object. Third, it gives provider objects, which perform the main functionality (e.g. sorting, searching, computing). These objects do not perform any system control, but provide certain functionality for aggregation and coordination objects.

In each phase, a human expert is needed. In the first phase, various ambiguities may arise, for example, to which particular object a procedure should be assigned. The second phase is implemented totally by a human expert. In the third phase, using

application domain knowledge, the human expert can perform the matching between reverse and forward model in ambiguous situations. Many parts of the fourth phase can be automated. However, the human expert has to deal with adapting the interfaces of procedures and with the decomposition of the procedural remainder. Especially, the author [12] considers the means to decrease human intervention. This kind of methods relies heavily on the domain analysis of the application to migrate. In addition to the cost of this analysis, which is usually very high, most of the legacy applications are not documented and the domain expertise is not always available, making this method of little practical use.

2.3.4 Other Methods to Identify Objects

There are a lot of other methods to identify objects from legacy code. Zimmer (1990) [58] outlines the need for restructuring FORTRAN programs to achieve a style related to data abstraction and object-oriented programming, and proposes a method for identifying objects by searching for cobwebs of related variables and mapping them on global invariants. Jacobson and Lindström [33] show how old systems can be gradually re-engineered to an object-oriented architecture. They consider different cases depending on whether the change concerns implementation technique or functionality. They discuss three different situations: a complete change of implementation technique and no change in functionality, a partial change in implementation technique and no change in functionality, and a change in functionality. If the implementation technique changes only partially, the new part (object-oriented) and the remaining part of the old system (non-object-oriented) must be made fit together. An interface is needed between these different parts via which they can communicate with each other. But this technique does not suggest a method for identifying potential objects in legacy systems.

Concept analysis has also recently been applied to the problem of identifying objects in legacy systems [38, 53, 50]. Concept analysis provides a way to identify groups of

entities that have common attributes. Each group constitutes a concept, while concept partitions are represented by collections of concepts that partition the set of entities (Siff and Reps, 1997). In the case object identification, routines are mapped onto entities, while attributes such as variable usage (uses global variable *v*) (Lindig and Snelting, 1997), or type usage (return type is *t*, has argument of type *t*, uses structure field of type *t*) (Siff and Reps, 1997) are considered; a concept partition represents a possible decomposition of a system into objects. The main advantage of this approach is the possibility of using negative attributes, such as "does not use structure fields of type *t*" (Siff and Reps, 1997), to obtain more refined objects. Two major limitations are the need to define the relevant attributes carefully and the high number of concept partitions resulting when the method is applied to large systems. Sahraoui et al. [50] apply concept analysis to the object reference view in order to detect abstract data objects. Their approach differs from Lindig and Snelting's approach by distinguishing different kinds of variable references and by the interpretation of the concept lattice. The identification of atomic components in the concept lattice is divided into three steps: first, identification of sets of variables; second, merging of overlapping sets of variables; third, identification of subprograms. The term object is used by Sahraoui et. al. in the meaning of object-oriented programming. This originates in the object-oriented approach. The definition of a concept requires of all routines to reference all variables and since variables are rarely referenced by the same set of routines in reality, concepts with a huge number of variables virtually do not exist. Thus, the concepts we can expect to find do not comprise a very large number of variables and concepts with many variables represent higher cohesion among the routines in the concept than concepts with a lower number of variables.

Philip and Ramsundar [48] use a dynamic method to get abstractions of function in order to re-engineer from a procedural language into an object-oriented one. They operate a system, invoke all its functions, and evaluate the behavior of the functions.

Based on the functionality, they get abstractions for each function. They analyze the corresponding code to refine the functions to lower levels. During this analysis, data elements are identified, and they are added to a data dictionary. To identify objects, the data dictionary is examined to identify items which correspond to specific components in the system. In most cases, these items become the objects in the target system. In addition, closely related items in the data dictionary are used as attributes of the found objects. After the objects are identified, they are compared against the remaining data items in the dictionary. Those items that describe an identified object are placed as attributes of that object. The algorithms of the functions refer to the data elements of the system. This information is used to assign functions as operations of the identified objects. Each function is assigned to an object, which closely matches the object or the attributes of the object.

Newcomb [45] describes a highly automated process to re-engineer procedural programs into object-oriented ones. His object-oriented model is based on state machines, and it is called hierarchical object-oriented state-machine model. Newcomb describes several analyses. Alias analysis, for example, examines records and their fields to find out the occurrences of records having a different name but an identical structure. The template for a record having different occurrences is called collision former. The records matching the collision former are called aliases. An alias map is a relation whose domain is a collision former and whose range is the set of alias records. He constructs control flow graphs, a data flow graph, and a state transition graph for procedures. The state transition graph consists of a start state, an end state and asset of intermediate states joined by state transitions. A state transition is defined by each distinct sequence of control conditions followed by a sequence of actions. The state transition table depicts one or more states and the conditions and actions involved in transitions between states. Newcomb derives classes in different ways. Data object classes, for example, can be obtained by the alias analysis, and a program object class is a class whose instances are top level programs.

Record types of procedural programs are usually converted into objects (classes) of object-oriented programs. However, this method does not suit `Cobol` programs very well. A record variable in `Cobol` may contain even about 40 fields. Thus, the corresponding classes would have too many attributes. Consequently, van Deursen and Kuipers [13, 14] have used cluster analysis to derive smaller objects. Cluster analysis generates a use matrix, telling which procedures use which variables. For each variable pair of that matrix, a Euclidian distance is calculated. The greater the distance is, the bigger the dissimilarity between the variables is. According to the distances, the variables sharing a great similarity can be grouped together.

2.4 Problems in Object Identification

As presented above, there are a lot of tools to aid object identification. However, usually the existing tools are not suitable for a certain situation. For example, they do not support the particular dialect of the programming language. Parsing-based tools may not accept a deficient program, for example, the program cannot be parsed if some included files are missing. In addition, usually the tools are neither flexible nor extendible.

A typical situation when re-engineering a legacy system is that the only existing document about the system is the source code. The other information about the system is in the heads of the designers and programmers. However, they may have left the company or moved to other duties. The lacking documents make re-engineering more difficult. Thus, many tools for re-engineering do not require any other document than code.

In object identification, domain knowledge is usually needed. Thus, totally automated tools do not yield good resulting code. The tools are typically semi-

automatic: routine work is performed automatically, but human assistance is needed in some decisions, for example in the decision whether to accept an object candidate as a final object. However, producing pure automated tools for as many re-engineering phases as possible is desirable, because they can significantly reduce resources needed for object identification.

Object identification area especially has its inherent difficulties in recovering objects in non-object-oriented code. When using graph-based solutions, for example, searching for isolated sub-graphs produces satisfactory results only for the ideal programs which have been designed according to a fully object based approach [5]. For other programs the sub-graphs are not totally isolated, and human assistance is needed to dissolve the insignificant relationships.

Chapter 3 *Conceptual Clustering*

The goal of object identification is to identify coherent groups of data items with those routines that use these data items in a given legacy software system. Conceptual clustering is a technique for finding related items in a data set. We apply conceptual clustering to the usage (grouping) of global variables throughout a legacy system, based on the hypothesis that global variables that are related in implementation (are used in the same routine) are also related in the application domain. In this chapter, we first give a brief introduction to conceptual clustering. Then we describe *Cobweb* system, a conceptual clustering technique, which our proposed object identification approach heavily relies on. We end the chapter with an example of *Cobweb* System for the four single-cell organisms to understand its algorithm well.

3.1 A Brief Introduction to Conceptual Clustering

Conceptual clustering is the grouping of a given set of instances (or observations) into conceptually simple clusters based on their attribute values. It was first introduced by Michalski [43]. Conceptual clustering is an unsupervised technique where the instances are presented without any bound to a particular concept. It

involves learning category structure from little previous information and using a domain-general (syntactic) similarity-based method to refine and construct new knowledge structure.

The traditional techniques developed in clustering analysis are often inadequate as they arrange instances into clusters solely on the basis of a numerical measure of instance similarity. The only information used is that contained in the instances themselves. Their algorithms are unable to take account of semantic relationships among instance attributes or global concepts that might be of relevance in forming a classification scheme. Consequently, the obtained clusters may have no simple conceptual description and may be difficult to interpret.

The traditional techniques' limitation is overcome by conceptual clustering, which arranges instances into clusters representing certain descriptive concepts rather than into clusters defined solely by a similarity metric defined initially in the attribute space. In brief summary as stated by Michalski and Stepp [44], conceptual clustering can be regarded as:

Given:

- A set of instances (physical or abstract);
- A set of attributes to be used to characterize the instances;
- A body of background knowledge, which includes the problem constraints, properties of attributes, and criteria for evaluating quality of constructed classifications;

Find:

- A hierarchy of object categories, in which each category is described by a single conjunctive concept. Subcategories that are descendant of any parent category should have logically disjoint descriptions and optimize an assumed criterion (a clustering quality criterion).

3.2 Cobweb System

In 1987, Fisher [16] presented *Cobweb* and since then it became one of the most used, studied and extended clustering systems. It has been used, with some variants in several domains like physics, design and medicine.

Cobweb is an incremental clustering system that generates a probabilistic categorization tree (hierarchical clustering with probabilistic described clusters). The number of clusters to be formed is automatically chosen and it has no parameters to be set. Instances, described by a list of attribute-value pairs, are presented to the system one by one and a hierarchical classification tree is formed. An instance is assumed to be a vector of nominal values V_{ij} along different attributes A_i. Cobweb employs probabilistic concept descriptions to represent the learned knowledge. In this sort of representation, in a category C_k, each attribute value has an associated conditional probability $p(A_i=V_{ij}|C_k)$ reflecting the proportion of instances in C_k with the value V_{ij} along the attribute A_i. Each category in the hierarchy includes probabilities of occurrence for all values of all attributes. This is essential to both categorizing new instances and modifying the category structure to better fit new instances. When given a new instance, *Cobweb* considers the overall quality of either placing the instance in an existing category or modifying the hierarchy to accommodate the instance. The criterion *Cobweb* uses for evaluating the quality of a classification is called *category utility* [28].

Category utility attempts to maximize both the probability that two instances in the same category have values in common and the probability that instances from different categories have different property values. Category utility is defined:

$$\sum_k \sum_i \sum_j p(A_i = V_{ij}) p(A_i = V_{ij} \mid C_k) p(C_k \mid A_i = V_{ij})$$

The sum is taken across all categories, C_k, all attributes, A_i, and all attribute values, V_{ij}. $p(A_i=V_{ij}|C_k)$, called predictability, is the probability that an instance has value V_{ij} for its attribute A_i, given that the instance belongs to category C_k. The higher this probability, the more likely the instances in category C_k share the same attribute value V_{ij}. $p(C_k|A_i=V_{ij})$, called *predictiveness*, is the probability with which an instance belongs to category C_k given that it has value V_{ij} for its attribute A_i. The greater this probability, the less likely instances not in the category will have the attribute value V_{ij}. $p(A_i=V_{ij})$ serves as a weight, assuring that frequently occurring attribute values will have stronger influence on the evaluation. By combining these values, high category utility measures indicate a high likelihood that instances in the same category will share properties, while decreasing the likelihood of instances in different categories having properties in common.

Using the Bayes' rule we have $p(A_i=V_{ij})p(C_k|A_i=V_{ij})=p(C_k)p(A_i=V_{ij}|C_k)$. Thus we can transform the above expression into an equivalent form:

$$\sum_k p(C_k)\sum_i\sum_j p(A_i=V_{ij}\mid C_k)^2$$

Gluck and Corter have shown that the sub-expression $\sum_i\sum_j p(A_i=V_{ij}|C_k)^2$ is the expected number of attribute values that one can correctly guess for an arbitrary member of category C_k. This expectation assumes a probability matching strategy, in which one guesses an attribute value with a probability equal to its probability of occurring. They define the category utility as the increase in the expected number of attribute values that can be correctly guessed without such knowledge. The latter term is $\sum_i\sum_j p(A_i=V_{ij})^2$, which is to be subtracted from the above expression. Thus the complete expression for the category utility the follow:

$$CU = \frac{\sum_k p(C_k)(\sum_i \sum_j p(A_i = V_{ij} \mid C_k)^2 - \sum_i \sum_j p(A_i = V_{ij})^2)}{k} \quad (3.1)$$

The difference between the two expected numbers is divided by k, which allows us to compare categories of different size.

The *Cobweb* algorithm is defined:

```
N = Node; I = New instance
Cobweb(N, I) =
   IF leaf(N)
  THEN create_subtree(N, I)
  ELSE
       Incorporate(I, N) ; Updates N's probabilities

        Compute score of placing I in each child of N
        N1 = child with highest score High
        N2 = child with second highest score
        New = score when placing I as a new child of N
        Merge = score for merging N1 and N2
        Split = score for splitting N1 into its children

        IF highest score is :
        High : THEN Cobweb( N1, I)
        New : THEN add I as a new child of N
        Merge : THEN Cobweb(merge(N1, N2, N), I)
        Split : THEN Cobweb(split(N1, N), I)
```

Cobweb performs a hill-climbing search of the space of possible taxonomies using category utility to evaluate and select possible categorization. It initializes the taxonomy to a single category whose attributes are those of the first instance. For each subsequent instance, the algorithm begins with the root category and moves through the tree. At each level it uses category utility to evaluate the taxonomies resulting from:

- Placing the instance in an existing category;
- Adding a new category and placing the instance into it;
- Combining two categories into a single category (merging);
- Dividing a category into several categories (splitting).

To merge two nodes, the algorithm creates a new node and makes the existing nodes children of that node. It computes the probabilities for the new node by combining the probabilities for the probabilities for children. Splitting replaces a node with its children.

This algorithm is efficient and produces taxonomies with a reasonable number of clusters. Because it allows probabilistic membership, its categories are flexible and robust. In addition, it has demonstrated base-level category effects and, through its notion of partial category matching, supports notions of prototypicality and degree of membership.

3.3 An Example of Cobweb

This section presents a sample concept hierarchy tree built by *Cobweb*. Figure 4 illustrates a *Cobweb* taxonomy taken from Gennari et al. (1989). In this example, the algorithm has formed a categorization of the four single-cell animals at the bottom of the figure. Each animal is defined by its value for the attributes: number of tails, color, and number of nuclei. The members of category C_3, for example, have a 1.0 probability of having 2 tails, a 0.5 probability of having light color, and a 1.0 probability of having 2 nuclei. As the figure shows, each category in the hierarchy

includes probabilities of occurrence for all values of all attributes. This is essential to both categorizing new instances and modifying the category structure to better fit new instances. If the clustering with the singleton category emerges as the winner, *Cobweb* creates this new category and makes it a child of the current parent node. Node C_6 in the figure was created in this fashion, since the instance it summarizes was sufficiently different from node C_2 and C_3.

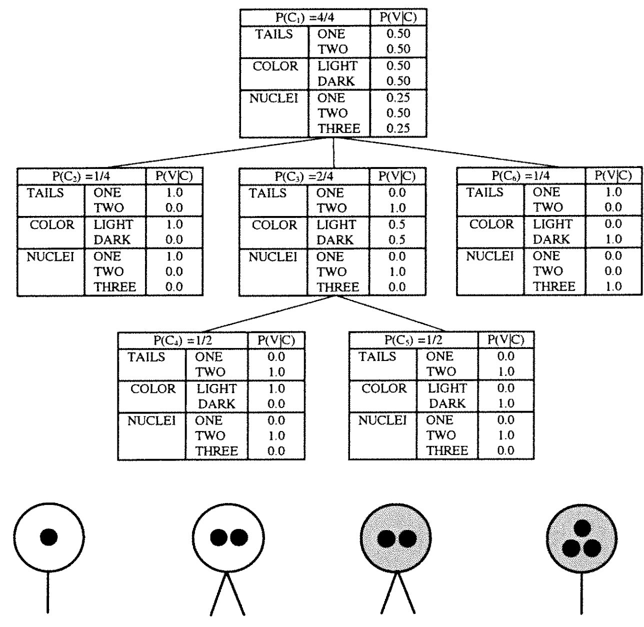


Figure 4 A Cobweb clustering for four one-celled organisms

3.4 Some Important Properties of Cobweb system

Cobweb system is able to afford the following advantageous properties, which are considered to be important and appropriate to our object identification problem. These promising properties are main reasons why our algorithm heavily relies on *Cobweb* system. We list them as follows:

Unsupervised

Learner has to cluster instances without advice from a teacher; in other words, learner must decide not only which instances each cluster should contain, but also the number of such clusters. In view of object identification problem, given the information directly extracted from a legacy system as its input, our algorithm is supposed to group closely and functionally related data items together automatically with little knowledge about the system in advance.

Incremental

It does not require that all instances be present before it begins learning. It could construct usable concept descriptions from an initial collection of data and update that description as more data become available. This feature provides the flexibility that allows our algorithm to deal with one subsystem alone, many subsystems together, and a whole system with various tasks. The flexibility thus enhances our algorithm performance.

Hierarchical

It constructs a concept hierarchy. The root represents the most general concept that summarizes the entire instances. The higher level nodes represent more general concepts than the lower level nodes. Each node in a concept hierarchy not only represents a concept, but also contains an intentional description of that concept. In view of the concepts, such as composition or inheritance in OO model, the hierarchy is able to provide some information about object-features in it and help our further understanding the program structure in a legacy system.

Chapter 4 Conceptual Clustering Based Approach

This chapter presents the conceptual clustering based approach. It first gives the overview of the object identification approach by Sahraoui [50], in which my work is embedded. It describes an example of applying *Cobweb* system to object identification to illustrate its limitations. Then we turn our attentions to mitigating or even eliminating those undesirable features the original *Cobweb* has. At the end of this chapter, we present an example of object identification in order to illustrate how this approach works.

4.1 Overview of the Proposed Approach

The object identification approach Sahraoui (1999) proposed consists of five steps (see Figure 5). First, it computes some metrics to determine the profile of the application at hand. This profile allows us to choose the appropriate program abstraction that we can use to identify objects. Then we identify objects using different algorithms. Third, we identify the methods of these objects. The fourth step consists of identifying the relationships between the objects (generalization, aggregation, or more generally, associations). Finally, the source code is transformed using the so-derived object model. In this thesis, we limit ourselves to application

profiling in first step and the second step. In remainder of the section, we briefly introduce them. A detailed description is given in the following sections.

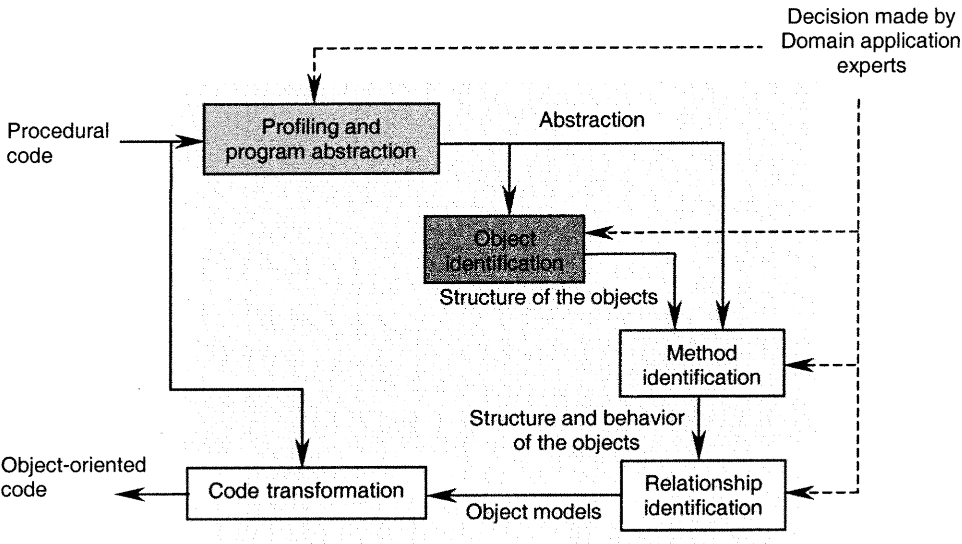


Figure 5 Overview of the object identification approach

Application profiling and program abstraction: We establish two sets of relations, one is the relationship between routines and data (for each data, which routines use it), the other is the relationship between routines (for each routine, which other routines are called by it). These two sets of relations are used as inputs of our algorithm.

Object identification: In an OO design, an application is modeled by a set of objects where objects are composed of a set of data and a set of operations that manipulate the data. Most of the graph based approaches to object identification group data with the routines that use the data. Our approach uses conceptual clustering based algorithm.

4.2 An Example of Applying Cobweb to Object Identification

In this section, an example of the direct usage of *Cobweb* to an imaginary small legacy system is given to explain how *Cobweb* is applied to the object identification and to offer some additional insight into the limitations of *Cobweb* to this problem.

The target system consists of 6 global variables (data items) and 11 routines as shown in Figure 1. It shows the reference relations between the routines r_i's and the global data d_i's of this system built by Sahraoui et al [50]. Figure 3 in Chapter 2 shows the corresponding reference graph, where nodes are either routines or global data, and an edge between a routine and a data means that the routine uses the data [15]. This kind of graph is used to identify objects, where each isolated sub-graph is considered to embody an object. Three candidate objects $\{d_2, d_4\}$, $\{d_5\}$, and $\{d_1, d_3, d_6\}$ can be obtained from the reference graph. Although the algorithm based on the search for notable sub-graph and /or patterns may, in many cases, produce low quality objects, or even more than one object within the same candidate, a good result can be obtained in the particular example since no routines that reference the data items of two objects create a link between the corresponding sub-graphs.

For our purpose of applying *Cobweb* to object identification, each data item can be a represented by a set of values in cells of each column in Figure 1, where a cell filled with value 1 stands for the routine references the data item and an empty cell stands for the routine does not reference the data item. Note that each data item contains a large proportion of the routines with value 0 (or empty cells in Figure 1), and the presence of these routines distorts the meaningful concept formation of the given set of the data items. To overcome this problem, it is necessary to introduce the tailored algorithm for missing values in [3]. Further, the algorithm redefines the CU measure so that comparison between a concept node and its parent is made only over the set of routines that are common to both nodes. Therefore, the term for the parent node in the CU calculation is summed only over $i \in$ child node. In other words, CU function

is defined to measure the increase in predictability only for those routines that have observed values (routine with value 1) in child node. Unobserved routines with value 0 are ignored.

By using the category utility CU as an object function, the clustering tree for object identification is built in Figure 6. Considering a single-level partitioning of the data items as candidate objects, two objects $\{d_1, d_3, d_6\}$ and $\{d_2, d_4, d_5\}$ are determined, which are different from those obtained from the above reference graph in the number of objects and their composition of data items in objects.

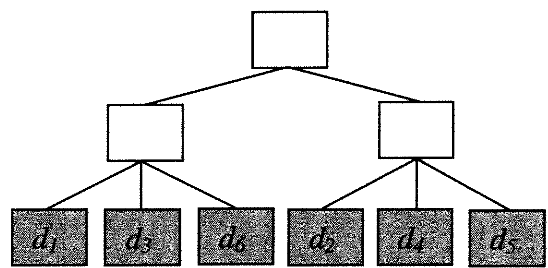


Figure 6 Clustering tree for object identification

As we know that the different techniques have their own different clustering criteria of identifying coherent groups of data items, the reference graph technique achieved a better recall for this system. Apparently, this system has a better decomposition. The direct usage of *Cobweb* to object identification, however, is based on a single value of the object function, which provides the tradeoff between maximal intra-cluster similarity and inter-cluster difference and favors making two groups, one with relatively high similarity of attributes, and the other with relatively low similarity of attributes among data items. In the following section, the limitations of *Cobweb* to object identification will be discussed.

4.3 Limitation of Applying Cobweb to Object Identification

Based on our understanding the conceptual clustering approach in Chapter 3 and the result of the above example, we point out the following limitations of applying *Cobweb* to object identification directly. These limitations and other concerns led us to propose an improved technique of object identification, which is more efficient and takes additional information extracted from a legacy system into account.

4.3.1 Cobweb Algorithm Itself

Its order dependency

One of undesirable characteristics of *Cobweb* is that it can be sensitive to the order of instance presentation, creating different hierarchies from different orders of the same data [42]. Although MERGE and SPLIT operators are introduced in *Cobweb*, these two operators can only mitigate the ordering problem locally but can not improve the whole nonoptimal hierarchies. In addition, these two operators' cost can be comparatively high.

Its hierarchy tree height

A hierarchical tree built by *Cobweb* can be grown to arbitrary height. As the tree gets deeper, the cost of the above phases grows substantially. In general, the higher level nodes in the tree represent more general concepts than the lower level nodes. Furthermore, the higher levels reflect meaningful structure in the data set more significantly than the lower levels. The resulting hierarchical tree can be bounded to some limited height, which, however, keeps sufficient information to interpret the induced results in view of object identification so as to reduce its cost.

4.3.2 Object Function CU

CU is the criterion *Cobweb* uses for evaluating the quality of a clustering. It stresses the tradeoff between maximum intra-cluster similarity and inter-cluster difference.

However, it may give a relatively high evaluating value when an instance is placed into one cluster with low similarity (cohesion) and very low dissimilarity (coupling) to the other clusters. Such circumstance raises the question whether the criterion is appropriate to object identification.

Furthermore, CU makes use of information that what routines access a global variable as shown in the above example. But in view of object identification, it is also of interest what other routines are called by a routine since it is closely relevant to the quality of object identification. A relatively high value of CU can be acquired if a global variable is put in a cluster with respect to its some common routines that access the variable. However, if some routines that access the variable call many other routines in other distinctive clusters, ignoring the calling relation between routines in different clusters may produce low quality objects. The calling relation may make undesirable high coupling between objects. Therefore, the relation must also be borne in mind. CU is unable to make use of the information according to its definition. One of our challenges in applying *Cobweb* to object identification is to define an appropriate object function.

4.4 Conceptual Clustering Based Approach

The section will present the conceptual clustering based approach for identifying objects. It first gives an overview of the approach. Then it discusses the various aspects in details, which are taken into account in the approach. We will finish this section with an example of applying the approach to a well-known small system in the literature.

4.4.1 Approach Overview

The approach heavily relies on *Cobweb* algorithm and introduces a process of iterative optimization to seek higher quality of objects decomposed from a legacy system. We propose a new algorithm, called *OI-Cobweb* that makes some

modifications on several aspects of *Cobweb*'s properties, where it creates new clustering criterion functions (objective functions). The criterion functions are based on object oriented design metrics (cohesion and coupling). *OI-Cobweb* makes use of the two relations extracted from a legacy system, one is the usage of each variable in a set of routines, the other is the call relation among routines, which renders information that what other routines are called or used by each routine. Multi-criteria decision-making was adopted in *OI-Cobweb* to prevent a candidate object from turning out to be a highly cohesive cluster while highly coupling with other objects or a low cohesive cluster while coupling with other objects. Such bad objects may be produced while using single criterion decision-making like CU in *Cobweb*. In addition, the approach puts a *Reordering* procedure into the iterative optimization process and sets the terminal condition of the process that minimizes the total couplings between candidate objects. The high quality of objects is obtained while trying to maximize the cohesion in each object and minimize the coupling between the objects.

Given a set of global variables arranged in a random order, the algorithm applies *OI-Cobweb* to build a conceptual clustering tree. In iterative process, the algorithm keeps the previously built clustering tree, applies the *Reordering* procedure to rearrange the global variables in an intended order ('dissimilar' order) as the next input and rebuild a new clustering tree. Then the algorithm compares the qualities of previous and new clustering trees according to their total sum of couplings between candidate objects. The process terminates when there is no further improvement in clustering quality. Each set of global variables covered by each first-level node in the tree is identified as a candidate object.

The approach algorithm is defined as the following where *OI-Cobweb* algorithm can be found in section 4.4.5 of this chapter.

Algorithm 1

Given $\{V_{initial}\}$, a set of global variables in a random order initially
Apply OI-Cobweb($\{V_{initial}\}$) to build $\{C_{initial}\}$, an initial clustering tree

Repeat
    Reordering($\{C_{initial}\}$) to get $\{V_{new}\}$, the set of variables in a new order
    Apply OI-Cobweb ($\{V_{new}\}$) to build $\{C_{new}\}$, a new clustering tree
    If (Sum-Coupling($\{C_{initial}\}$) < Sum-Coupling($\{C_{new}\}$))
        keep $\{C_{new}\}$ as an initial clustering tree, Continue Repeat
    Else keep $\{C_{initial}\}$ as final clustering tree, Stop Repeat

4.4.2 Two Relations

Our proposed approach presumes that data and routines that are related in implementation in procedural software systems are also related in the application domain. A set of binary relations between the global variables and the routines is necessarily established, which indicates that the routines somehow manipulate the variables. We call the set of relations ***VR-relation***. However, most legacy systems are sequential and monolithic. Multiple tasks are performed within a routine by calling other routines. Routine call is the most primitive and dominating type of connector of such systems and may capture the additional useful information that global variables are related to each other frequently via such routine call. The set of relations, that a routine call a set of other routines, is termed ***RR-relation***. The two relations, extracted from a legacy system, are supposed to significantly capture the basic structural information leveraged for object identification. The following gives their definitions.

Definitions. Let a legacy system consist of a set of routines R and a set of global variables V. *VR-relation* is defined as $VR \subseteq V \times R$. If $(v, r) \in VR$, routine r uses

global variable v. *RR-relation* defined as $RR \subseteq R \times R$. If $(r_i, r_j) \in RR$, where $i \neq j$, routine r_i calls routine r_j.

Figure 7 shows an imaginary system, which consists of a set of routines $\{r_i\}$ and a set of global variables $\{v_i\}$ *VR-relation* and *RR-relation* of this system can be established, which is shown in Figures 8 and 9.

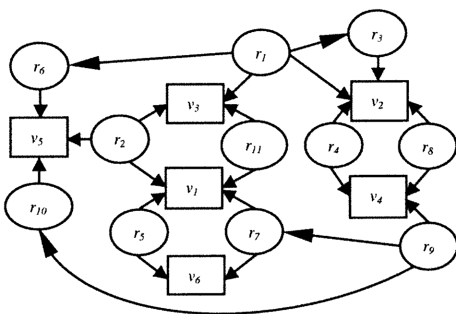


Figure 7 An Imaginary System

VR	v_1	v_2	v_3	v_4	v_5	v_6
r_1		1	1			
r_2	1		1		1	
r_3		1				
r_4		1		1		
r_5	1					1
r_6					1	
r_7	1					1
r_8		1		1		
r_9				1		
r_{10}					1	
r_{11}	1		1			

Figure 8 VR-relation from An Imaginary System

RR	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}	r_{11}
r_1			1			1					
r_2											
r_3											
r_4											
r_5											
r_6											
r_7											
r_8											
r_9							1			1	
r_{10}											
r_{11}											

Figure 9 RR-relation from An Imaginary System

Another example of *VR-relation* and *RR-relation*, which are extracted from a mini system, called *Collections*, can be found in Figures 11 and 12 respectively in section 4.4.7 of this chapter.

4.4.3 Clustering Criterion Functions of OI-Cobweb

Cobweb uses Category Utility (CU) as its clustering criterion function to evaluate the quality of a classification. Clustering criteria here adopted by *OI-Cobweb* consist of two criterion functions, one termed *Cohesion* that evaluates the cohesion in an object (a set of global variables), the other termed *Coupling* that evaluates coupling of an object with other objects. Both *Cohesion* and *Coupling* are inspired by those existing well-known design cohesion and coupling measures for object-oriented systems. *Cohesion* directly uses ***Coh*** [4], which is a variation of LCOM5 [52] and *Coupling* borrows the definition of ***COB*** [9] and counts the frequencies of links an object has with other objects. We cannot ignore the actual number of dependencies existing among objects, i.e., objects with only one dependency have the same *Coupling* as

objects with hundreds of dependencies. Both of them can be precisely defined as follow.

Let $V = \{v_1, v_2 \ldots v_n\}$ be a set of n variables; $R = \{r_1, r_2, \ldots r_m\}$ be a set of m routines The elements of V are grouped into a set of k clusters $C = \{C_1, \ldots C_i, \ldots C_j, \ldots C_k\}$ where an element $C_i = V_i \subseteq V$.

Cohesion of cluster C_i

The cohesion of a group C_i is defined as the number of routine-variable references over the number of all possible combinations routine-variable. Formally

$$Cohesion\ (C_i) = \frac{\sum_{k=1}^{n'} \mu(v_k)}{m' * n'} \tag{4.1}$$

where n' is the number of variables in C_i ($\#V_i$), m' is the number of routines that use the variables of V_i and $\mu(v_k)$ is the number of routines that use a variable v_k.

Coupling of cluster C_i to the other clusters C-C_i

The coupling of a group C_i with the other groups of the system C - C_i is defined as the number of external references of C_i. Formally

$$Coupling(C_i, C - C_i) = \sum_j (NRV(C_i, C_j) + NRR(C_i, C_j)) \tag{4.2}$$

where $C_j \in C$, $i \neq j$, $NRV(C_i, C_j)$ is the number of occurrences that the routines in C_i use the variables in C_j and $NRR(C_i, C_j)$ is the number of occurrences that the routines in C_i use the routines in C_j and inversely the routines in C_j use the routines in C_i.

A scenario of three clusters, C_1, C_2, and C_3 from the imaginary system, introduced above in Figure 7 is shown in Figure 10 and an example of their calculations of *Cohesion* in cluster C_2 and *Coupling* of C_2 with the other clusters is given out.

$$Cohesion(C_2) = \frac{\sum_1^{n'} \mu(v)}{m'*n'} = \frac{4+3+2}{5*3} = 0.6$$

$$\begin{aligned} Coupling\ (C_2, \{C_1, C_3\}) &= (NRV\ (C_2, C_1) + NRR\ (C_2, C_1)) \\ &+ (NRV\ (C_2, C_3) + NRR\ (C_2, C_3)) \\ &= (1+1) + (1+2) = 5 \end{aligned}$$

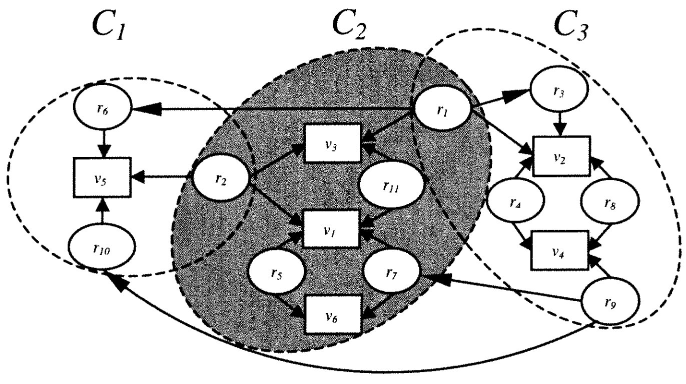


Figure 10 A Scenario of Three Clusters

4.4.4 Multi-Criteria Decision Making

In the same way as *Cobweb* performs, for each subsequent global variable, *OI-Cobweb* begins with the clustering tree root and moves through the tree. At each level it uses two criterion functions to evaluate the overall quality of either placing the variable in an existing category (an existing node in the tree) or adding a new category (creating a new node in the tree) and placing the variable into it. As

described above, the two criteria functions are associated directly to the selected object-oriented design metrics, cohesion and coupling. For each of the identified objects, C_i, its quality may be considered good if a high value of cohesion in it is achieved by *Cohesion*(C_i) and a value of its coupling with other objects by *Coupling*(C_i, C-C_i) turn out to be low.

Ideally, objects should be internally cohesive, i.e. the variables in an object should be kept together, and meanwhile inter-dependency between the objects should be kept as loose as possible. However, in the process of identifying objects using the incremental algorithm, we have to frequently face two options: to place a variable into a certain object resulting in a strong cohesion in it and also a high degree of coupling with other objects or into another object which makes the resulting object have a low cohesion while it loosely couples with others. Such situations indicate that object identification sometimes trades cohesion in an object against its coupling with others, and vice verse.

Considering the circumstances, our approach introduces multi-criteria decision making into the process of building clustering tree incrementally. It sets the prerequisite that when a variable is placed into a cluster, the resulting cluster (a candidate object) C_i should acquire an accepted degree of cohesion in it, i.e. a value achieved by *Cohesion*(C_i) no less than a predefined threshold value, which will be discussed in detail in Chapter 5 and 6. When the variable is placed into these existing clusters respectively, those potential clusters that satisfy the prerequisite are determined. Then among these potential clusters, the approach searches for the cluster that minimizes its coupling with other clusters. If there does not exist any such cluster that satisfies the prerequisite, a new cluster will be created for the variable.

As we know, different systems have different levels of quality, such as modularity, information hiding, and usage of work variables. The multi-criteria decision making can also provide our approach provide a flexible management means to different migration systems by setting cohesion threshold value. This advantage can be seen in the case studies of different systems in Chapter 6.

The multi-criteria decision making can be described precisely as follows.
Let $C = \{ C_1, \ldots C_i, \ldots C_j, \ldots C_k \}$ be the set of existing clusters and v_i a new coming variable to be clustered. v_i is accessed by a set of routines R_i; let C_i be a cluster. If v_i is placed into C_i, we will call the obtained cluster C'_i. We call the cohesion threshold value COHESION (a value ranging from 0 to 1). So whether cluster C_i the variable v_i is clustered into or placed into a new cluster C_{k+1} depends on the following.

$\forall C_i$ in C, Place v_i into C_i individually and its updated C'_i

IF $Cohesion(C'_i) <$ COHESION

create a new cluster C_{k+1} and place v_i into C_{k+1}, so updated clusters $C' = \{C_1 \ldots C_i, \ldots C_j, \ldots C_k, C_{k+1}\}$

ELSE

collect all clusters satisfying $Cohesion(C'_i) \geq$ COHESION;

place v_i into C_i **if and only if** MIN($\{Coupling(C'_i, C\text{-}C_i) \mid Cohesion(C'_i) \geq$ COHESION, $i = 1, \ldots k, \}$);

updated clusters $C' = \{C_1 \ldots C'_i, \ldots C_j, \ldots C_k\}$

The desirable characteristics of the multi-criteria making are that it insures all identified objects with a relatively high degree of cohesion and then keeps its

coupling with others as loose as possible, and offers the flexibility to our approach applied to different systems.

4.4.5 OI-Cobweb Algorithm

OI-Cobweb is inspired by the original *Cobweb*. It adopts its own criteria functions that directly link to object-oriented design measures, and the multi-criterion making as described and discussed above. Furthermore, it intends to make use of two kinds of relations, extracted from a legacy system, which are supposed to significantly capture the basic structural information leveraged for object identification. We believe that these modifications make *OI-Cobweb* more appropriate to object identification problem.

OI-Cobweb performs roughly the same clustering procedure as *Cobweb*'s. To further reduce the cost in the phrase of building clustering tree, *OI-Cobweb* does not use merging and splitting operations in *Cobweb*, whose computations are comparatively expensive. The roles these two operators play in mitigating order effects will be taken over by introduction of iterative process and the reordering procedure to our approach that are expected to mitigate ordering effects globally to a great extent.

Given an instance and a current partition, *OI-Cobweb* evaluates the quality of new clustering that results from placing the instance in each of the existing clusters, and the quality of the clustering that results from creating a new cluster that only covers the new instance; the option that yields the highest quality score based on multi-criteria making is selected. The clustering grows incrementally as new instances are added.

The following is the basic *OI-Cobweb* algorithm:

Algorithm 2

```
Given N = Node ; I = New instance
OI-Cobweb(N, I) =
  IF level(N) = TREE_HEIGHT –2 or
                leaf(N) and  level(N) < TREE_HEIGHT –2
    create_subtree(N, I)
  ELSE
    Incorporate(I, N) ; Updates N's database
    Compute cohesion of placing I in each child of N
    IF Cohesion(Ni) >= COHESION Threshold and Coupling(N1, N-Ni) is
      lowest
       Cobweb(Ni, I)
    ELSE
      create_subtree( N, I)
```

4.4.6 Reordering Procedure

A well-studied characteristic of incremental algorithms is their order dependency. They generate different clustering trees for different data orders. Although split and merge operators in *Cobweb* system may mitigate order effects, both operators are applied locally with each partition and order dependency could still affect the clustering tree structure.

Biswas's works [3] reveal that interleaved orders produce better clustering trees and stable final groupings in term of the rediscovery task. The "interleaved" corresponds to an order in which instances from different clusters are presented in sequence in an attempt to obtain a maximally dissimilar ordering among the instances. Taking this viewpoint, we presume that instances (global variables) that appear dissimilar tend to be placed in different clusters using the criterion functions in *IO-Cobweb* and

propose the following reordering procedure in order to extract the so-called interleaved order. It is expected that the introduction of reordering procedure to our approach may mitigate ordering effects globally and uncover better clustering to a great extent. The following function outlines the reordering procedure.

Algorithm 3

ORDERING(Root)
  IF Root is a leaf
  THEN Return(an instance covered by Root)
  ELSE
    Order children of Root from those covering the most instance s to those covering the least.
    For each child, C_k of Root (in order) Do $L_k \leftarrow$ORDERING(C_k)
      $L \leftarrow$MERGE({L_k /list of variables constructed by ORDERING(C_k)}})
      Return (L)

It recursively extracts a list of variables from the largest cluster to the least cluster, and then merges (i.e., interleaves) these lists, before exiting each recursive call -- at each step, an element from the largest cluster is placed first, followed by an element of the second largest, and so forth. Thus, this procedure returns a measure-dependent dissimilarity ordering by placing variables from different clusters back-to-back. Following initial clustering tree, Algorithm 3 extracts a dissimilarity ordering, reclusters, and iterates, until there is no further improvement in clustering quality.

4.4.7 An Example of the Approach

The following example, to which we apply the approach in identifying objects, is the well-known one introduced in Canfora et al (1996) -- called it *collections*. This example has the advantage of being self-contained, well known in the literature,

small, and yet relatively complex. Also, we add imaginary call relations to the example based on our understanding its implementation. This example presents a part of a C program shown in Figure 11. The program manipulates a stack, a queue and a list. In Figure 11, the body of each routine is replaced by a comment that indicates the list of global variables used by the routine and a list of routines called by the routine. The following demonstrates an example of object identification in order to illustrate this approach and its algorithm.

We establish two sets of relations, *VR-relation* and *RR-relation* from the program. Figures 12 and 13 show the matrix representations of a set of binary relations between global variables and routines, *VR-relation*, and a set of binary relations between routines, *RR-relation* in the program respectively. For the sake of readability, names of routines and variables are replaced by codes (number for a routines and letter for a variable). In Figure 12, we display a variable name and its corresponding letter in each cell of the first row and a routine name and its corresponding number in each cell of the first column. Using the approach, we build its clustering tree with 7 variables based on *VR-relation* and *RR-relation*. COHESION threshold value 0.65 and clustering tree height 4 are set for the example.

The following describes in a little more details how our approach works in identifying objects so that we can learn clearly and intuitively that such advantageous features as iterative optimization process, multiple criteria decision making, reordering procedure, which the approach acquire, contribute to uncovering better objects.

For this mini system with 7 global variables and 20 routines, the approach iterates three times arriving at its best result. We display how the approach builds its resulting clustering tree incrementally when given a order of the variables for each iteration, as shown in Figures 14, 15, and 16.

```
#define MAXIM 99
typedef int ELEM_T;
typedef int BOOL;
ELEM_T stack_struct[MAXIM];
int stack_point;
ELEM_T queue_struct[MAXIM];
int queue_head, queue_tail, queue_num_elem;
struct list_struct {ELEM_ node_content;
            struct list_struct * next_node;
            }  list;
main()
{
/* this program exploits a stack, a queue, and a list of items of type */
}

/* list of functions with as comment the list of global variables
referenced */
void stack_push(el)      {/* stack_point and stack_struct */
                          stack_full(), stack_top() }
ELEM_T stack_pop()       {/* stack_point and stack_struct */
                          stack_top(), stack_Empty() }
ELEM_T stack_top()       {/* stack_point and stack_struct */}
BOOL stack_Empty()       {/* stack_point */}
BOOL stack_full()        {/* stack_point */}
Void queue_insert()      {/* queue_struct, queue_head and queue_num_elem
                          */ queue_full() }
ELEM_T queue_extract()   {/* queue_struct, queue_tail and
                         queue_num_elem */ queue_Empty() }
BOOL queue_Empty()       {/* queue_num_elem */}
BOOL queue_full()        {/* queue_num_elem */}
Void list_add(el)        {/* list* / list_is_in() }
Void list_elim(el)       {/* list */ list_is_in() }
BOOL list_is_in()        {/* list */}
BOOL list_empty()        {/* list */}
Void global_init()       {/* stack_point, list, queue_head,
                         queue_tail and queue_num_elem */}
void stack_to_list()     {/* stack_point, stack_struct
                         and list */ stack_Empty(), list_add() }
void stack_to_queue()    {/* stak_point, stack_struct, queue_struct,
                         queue_head and queue_num_elem */ stack_Empty(),
                         queue_insert(), queue_full() }
void queue_to_stack()    {/* queue_struct, queue_tail,queue_num_elem,
                         stack_point and stack_struct */
                         queue_Empty(), stack_push(), stack_full() }
void queue_to_list()     {/* queue_struct, queue_tail,
                         queue_num_elem, and list */
                         queue_Empty(), list_add() }
void list_to_stack()     {/* list, stack_point, and stack_struct */
                         list_empty(), stack_push(), stack_full() }
void list_to_queue()     {/* list, queue_struct, queue_head
                         and queue_num_elem */
           list_empty(), queue_insert(), queue_full() }
```

Figure 11 Procedural Code of Collections in C

VR	a. stack_struct	b. stack_point	c. list	d. queue_tail	e. queue_head	f. queue_struct	g. queue_num_elem
1. stack_push	1	1					
2. stack_pop	1	1					
3. stack_top	1	1					
4. stack_empty		1					
5. stack_full		1					
6. stack_to_queue	1	1			1	1	1
7. global_init		1	1	1	1		1
8. list_is_in			1				
9. list_empty			1				
10. stack_to_list	1	1	1				
11. list_to_stack	1	1	1				
12. list_add			1				
13. list_elim			1				
14.queue_to_stack	1	1		1		1	1
15. queue_extract				1		1	1
16. queue_full							1
17. queue_empty							1
18. queue_insert					1	1	1
19. list_to_queue			1		1	1	1
20. queue_to_list			1	1		1	1

Figure 12 Matrix representation of a set of relations between routines and variable

Initially, the variables are presented in the following random order: ***a-b-c-d-e-f-g*** in Figure 14. In fact, this ordering is said to be 'similarity' one [3], that is, the worst case for uncovering good quality of object identification. Generally, 'similarity' orderings lead to poor clustering and 'dissimilarity' orderings lead to good clustering.

When given the first variable ***a***, a tree with only one node is built with the variable. The node is not only the root but also the leaf of the tree. As variable ***b*** comes, *OI_Cobweb* creates a new node and places ***b*** into it with the same parent node as ***a***'s showed in Figure 14(2). Note that actually variables ***a*** and ***b*** contain a large proportion of the common routines that access both ***a*** and ***b***, and should be clustered into same category. However they are separated into different clusters. The resulting

inappropriate clustering derives from the given 'similarity' order of ***a-b-c-d-e-f-g***. As we can see afterwards in figures 15 and 16, they will be clustered into the same category correctly. As a result of clustering in figures 15 and 16, it indicates clearly the importance of introduction of *iterative optimization process* as well as *Reordering* procedure into our approach.

RR	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1. stack_push			1		1															
2. stack_pop			1	1																
3. stack_top																				
4. stack_empty																				
5. stack_full																				
6. stack_to_queue				1												1		1		
7. global_init																				
8. list_is_in																				
9. list_empty																				
10. stack_to_list				1								1								
11. list_to_stack	1				1				1											
12. list_add								1												
13. list_elim								1												
14.queue_to_stack	1				1												1			
15. queue_extract																	1			
16. queue_full																				
17. queue_empty																				
18. queue_insert																1				
19. list_to_queue									1							1		1		
20. queue_to_list												1					1			

Figure 13 Matrix representation of a set of relations between routines

When variable ***c*** is fed in, there could be three clustering possibilities: to place ***c*** in an existing category, the same one as ***a***'s or ***b***'s, or to modify the tree hierarchy to accommodate variable ***c***. Using its criterion functions, *OI_Cobweb* evaluates the quality of first two possibilities respectively. Criterion function *Cohesion* turns out to give cohesion 0.57 for the cluster ***a*** and ***c***, and cohesion 0.59 for the cluster ***b*** and ***c***,

which are less than Cohesion threshold value 0.65 we set above, so that *OI_Cobweb* creates a new node for holding ***c*** according to multi-criteria decision making.

For variables ***d*** and ***e***, in the situation similar to ***c***'s, *OI_Cobweb* modifies the existing tree hierarchy to accommodate them separately (see (4) and (5) in Figure 14).

When given variable ***f***, *OI_Cobweb* places ***f*** into these existing clusters, ***a***, ***b***, ***c***, ***d***, ***e***, and evaluates their clustering quality using the criterion function *Cohesion* respectively. It finds out two potential clusters: one is to place ***f*** into cluster ***d***, the other is to place ***f*** into cluster ***e***, whose cohesion value are both 0.71, higher than the predefined Cohesion threshold value. Furthermore, in order to determine which cluster ***f*** will be placed into, *OI_Cobweb* evaluates their clustering quality of the two potential clusters in the aspect of its coupling with other clusters using the other criterion function *Coupling*. As a result of its evaluation, the cluster with variables ***f*** and ***e*** acquires lower coupling value 11 than that the cluster with variables ***f*** and ***d*** 15. The corresponding clustering tree is shown in Figure 14(6).

As the last variable ***g*** emerges, *OI-Cobweb* begins with the root node, evaluates categorization and selects one of the root's children in the same fashion as the above. The cluster selected among existing clusters is a parent node of ***e*** and ***f***. At the current level, *OI-Cobweb* extends downward and further finds the leaf node with ***f*** sharing a great proportion of properties with ***g***. Thus *OI-Cobweb* makes ***f*** and ***g*** the children of the current parent node. As a result, it builds the initial tree with *SUM-Coupling* 37.

Next, The approach start performing its iterative optimization until terminating condition is reached. A new order of the variables, ***f-a-b-c-d-e-g,*** from the above initial tree is extracted.. In the first iteration, the tree with *SUM-Coupling 28*, lower

than previous one is established. Figure 15 shows the whole process of building its tree.

Continually, extracting a new order, ***e-a-c-d-b-f-g*** from the tree in Figure 15, the new tree built in the second iteration with the same *SUM-Coupling* score as the one in the first iteration is obtained, shown in Figure 16. It indicates that there is no further improvement in overall quality of clustering tree. Thus, iterative optimization terminates.

It is quite interesting to note that *OI-Cobweb* builds the same clustering tree hierarchy although different orders of the variables are presented to it in the first and the second iteration respectively. Such result indicates that the 'dissimilarity' order *OI-Cobweb* is expected to have for better clustering is not strict. In fact, *OI-Cobweb* acquires such power to find a best clustering in few iterations. Figure 17 summarizes the results in the iterative optimization process.

We take the tree built in the first iteration with the variables presented in the order ***f-a-b-c-d-e-g*** as our final tree for object identification. We take the direct children nodes of the tree root, which contain variables, as the objects. In the example, we arrive at the following objects:

Object1=C_1={***d***, {***e***,***f***, ***g***}}

={queue_tail, queue_head,{queue_struct, queue_num_elem}}

Object 2=C_2={***c***}={list}

Object 3=C_3={***a***, ***b***}={stack_struct, stack_point}.

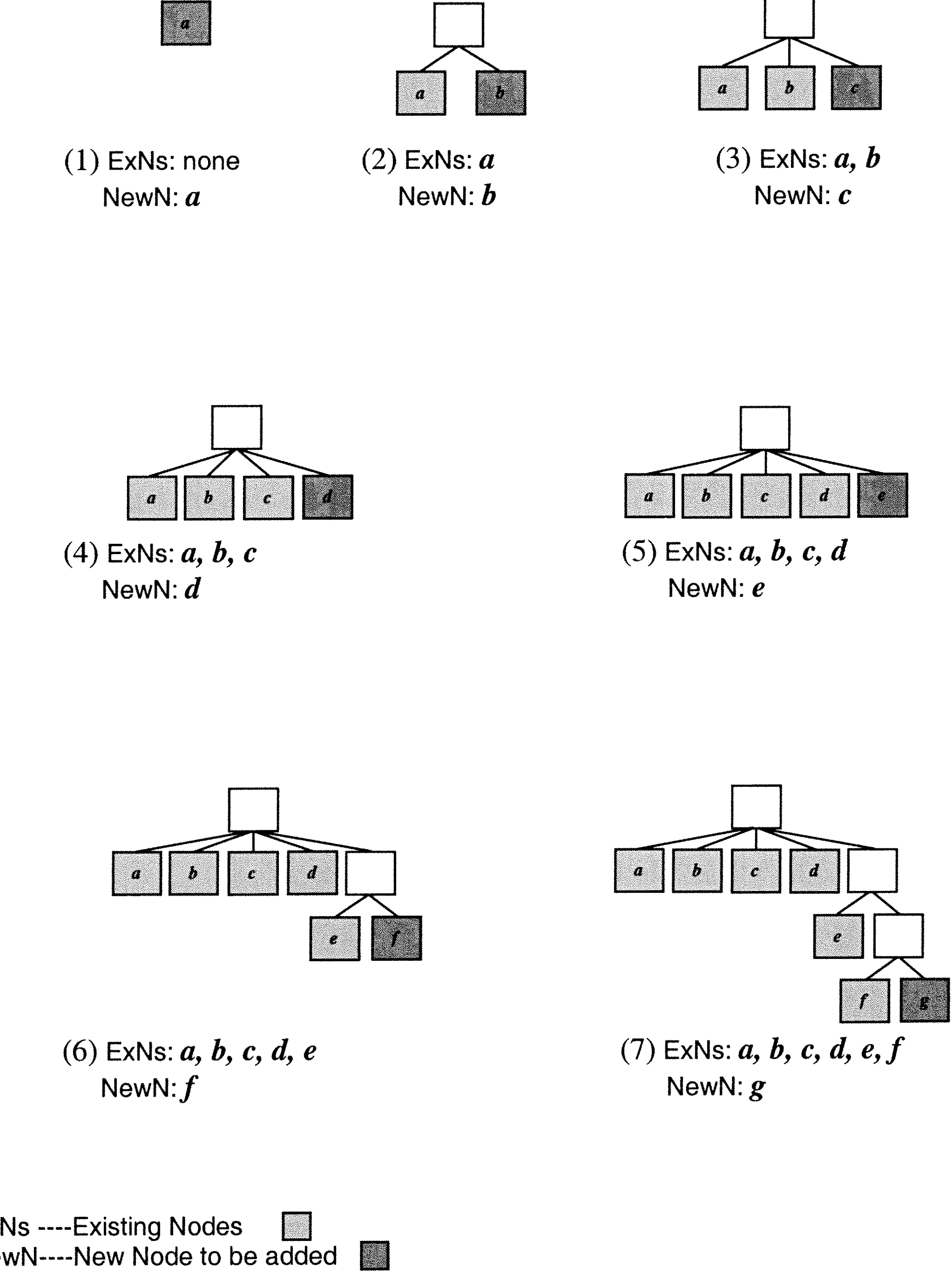


Figure 14 Initial Clustering of Variables in A Random Order

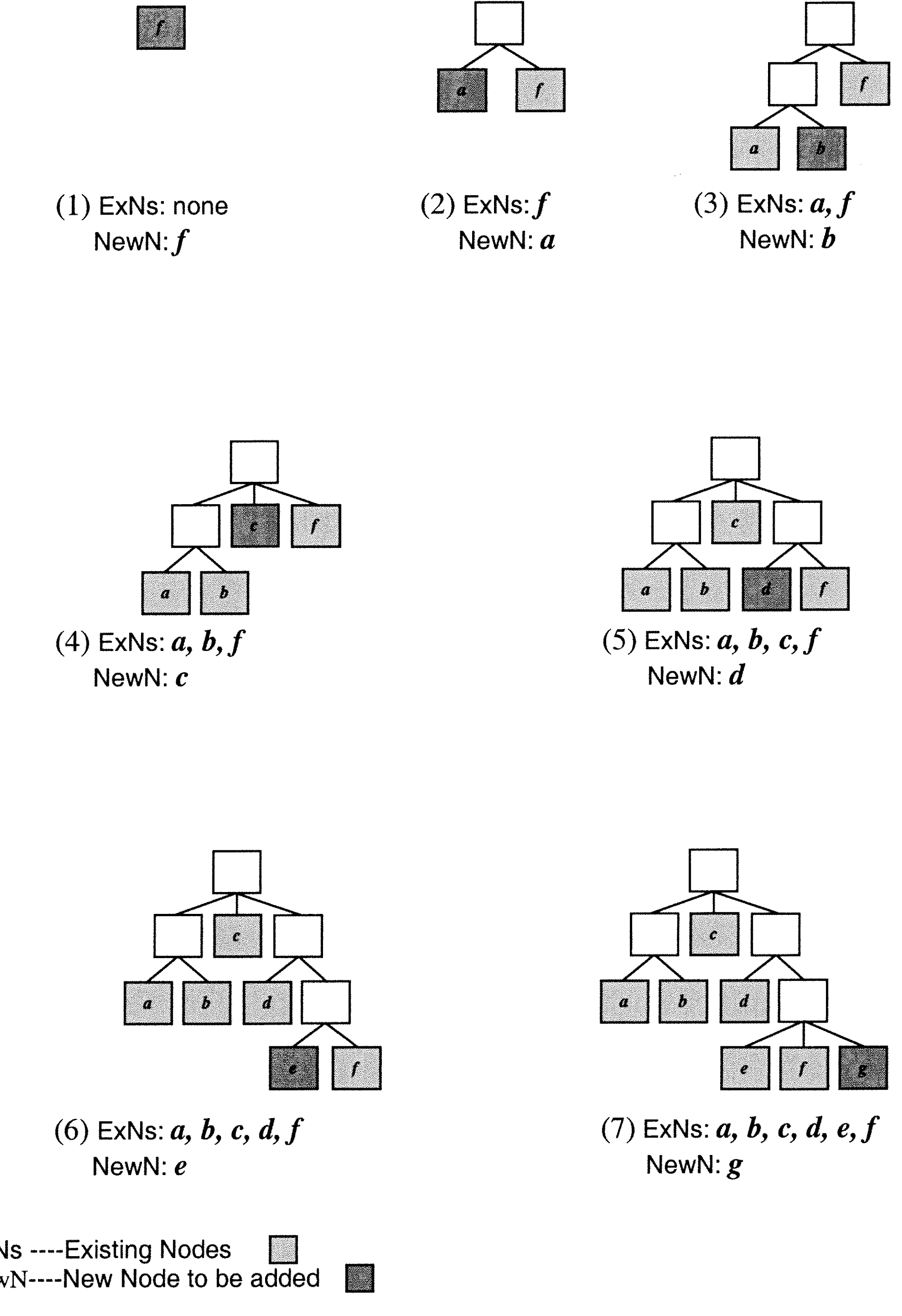


Figure 15 Clustering of Variables in A New Order in the Second Iteration

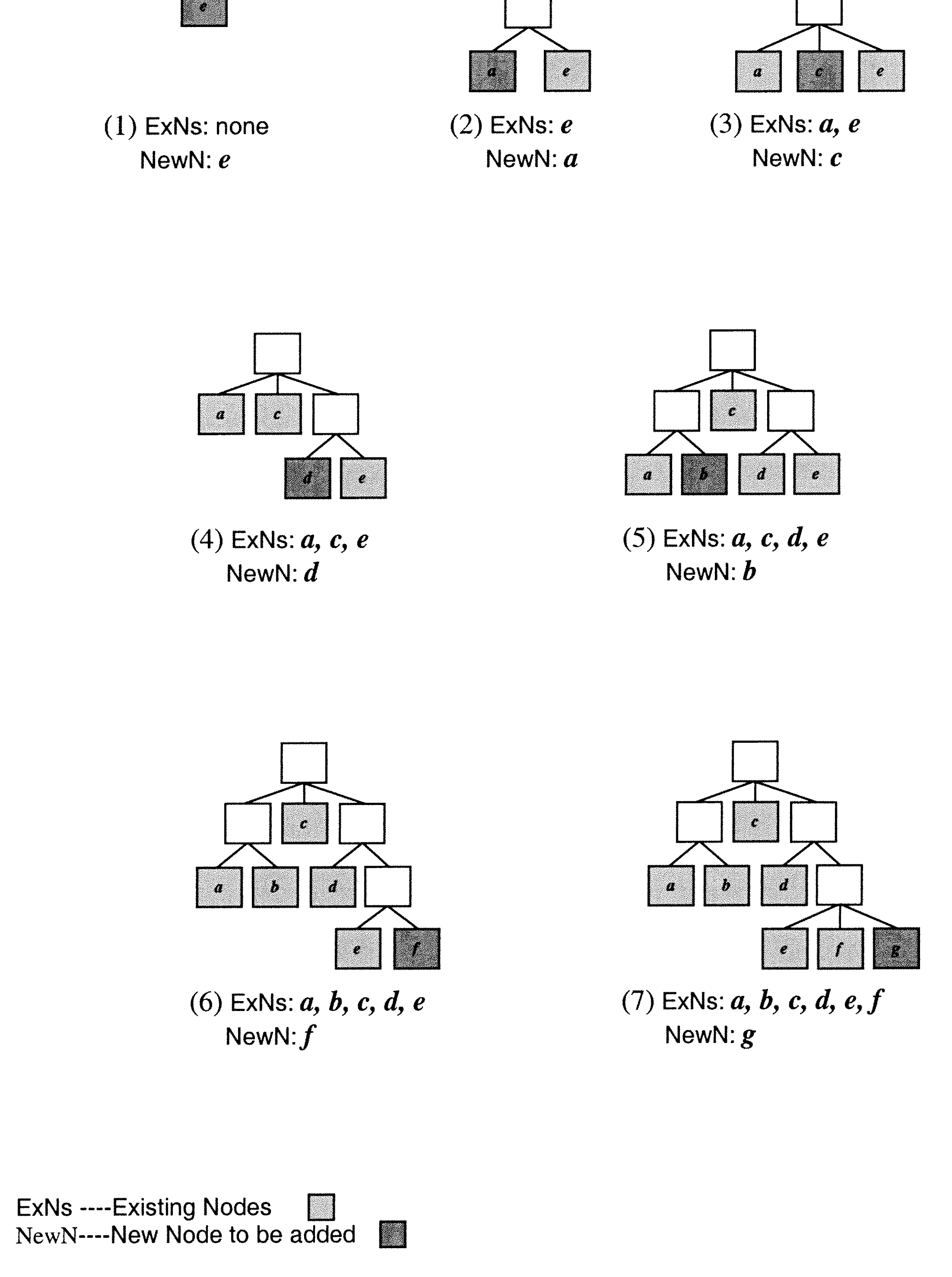
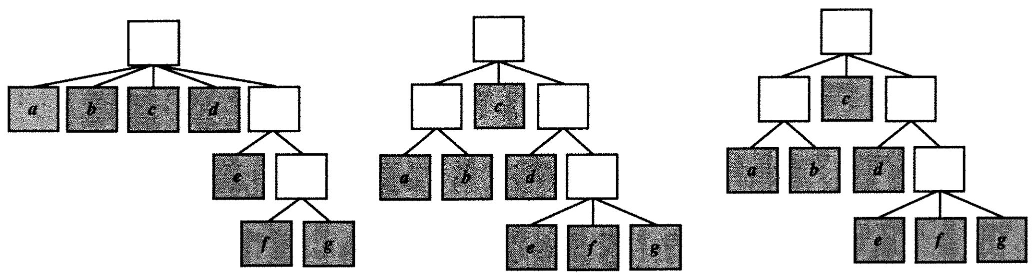


Figure 16 Clustering of Variables in A New Order in the Third Iteration



(a) ***a-b-c-d-e-f-g*** (b) ***f-a-b-c-d-e-g*** (c) ***e-a-c-d-b-f-g***

Figure 17 Different Clustering in different presented ordering

Chapter 5 *Implementation*

This chapter presents the prototype of our approach. First, a brief description of a supporting tool, DISCOVER's DeveloperXpress, which we use for the purpose of analyzing source code and extracting information as input for our approach, is given. Then we present some aspects of base facilities the prototype provides and how it is used for object identification.

5.1 Supporting Tool

DISCOVER's DeveloperXpress is a commercial tool that provides powerful navigation and query capabilities of existing software source code structure. It allows software developers to quickly find their way through code and to quickly understand a target software system. It supports many programming languages, such as Java, ANSI C/C++, K&R C, and a variety of SQLs on various operating systems, like Solaris, IRIX, WIN 32 and is integrated with other tools, such as ClearCase, SourceSafe, Emacs, Microsoft Visual Studio.

It is a parsing-based system that collects information about the relationships between language structures. It saves information about object domains, such as files, macros, data types, global variables and functions. Each object domain has attributes. For example a function has the following attributes: the name of the file containing it, its (return) type, its name, static variables used in it, its beginning and ending lines, its call expression, etc.. The relationships between each object domain are stored in the database. Thus, the normal database queries can be made. For example, the following queries are possible:

- Which functions refer to a certain global variable and a certain data type?
- Which functions are called by a certain function?
- Which functions take a certain global variable as their actual parameter?

Figure 18 shows an example of functions organized in DevelperXpress Brower.

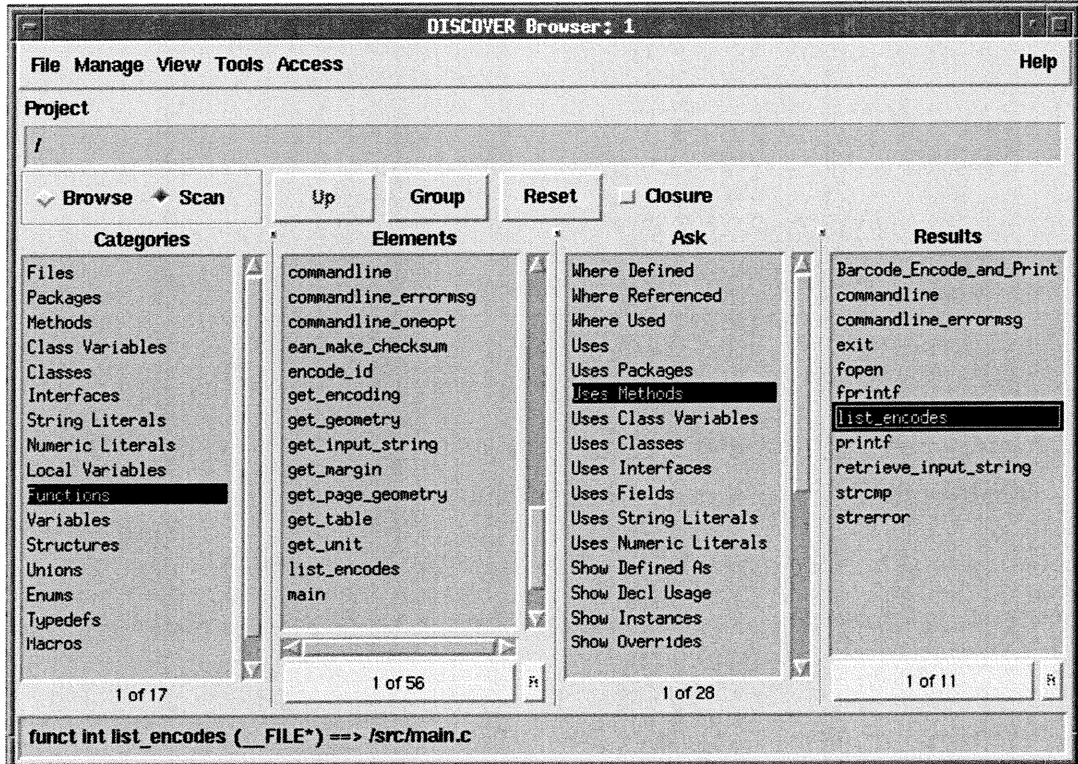


Figure 18 An Example in Brower of Discover's DeveoperXpress

It also enables one to analyze code through advanced graphical views to visualizing the software structures and the information of the resource flow graph of a software system. These structures and information include software components such as subsystems, calls, data accesses, interfaces, dependencies among components and data-flow relations. So we can define program segments to consist of statements, which are semantically related, but not necessarily physically adjacent. With the tool, these kinds of program segments can be isolated from other program text to examine them more precisely. Figure 19 shows an example of call-relation graph in the Viewer of `Discover`'s `DeveoperXpress`.

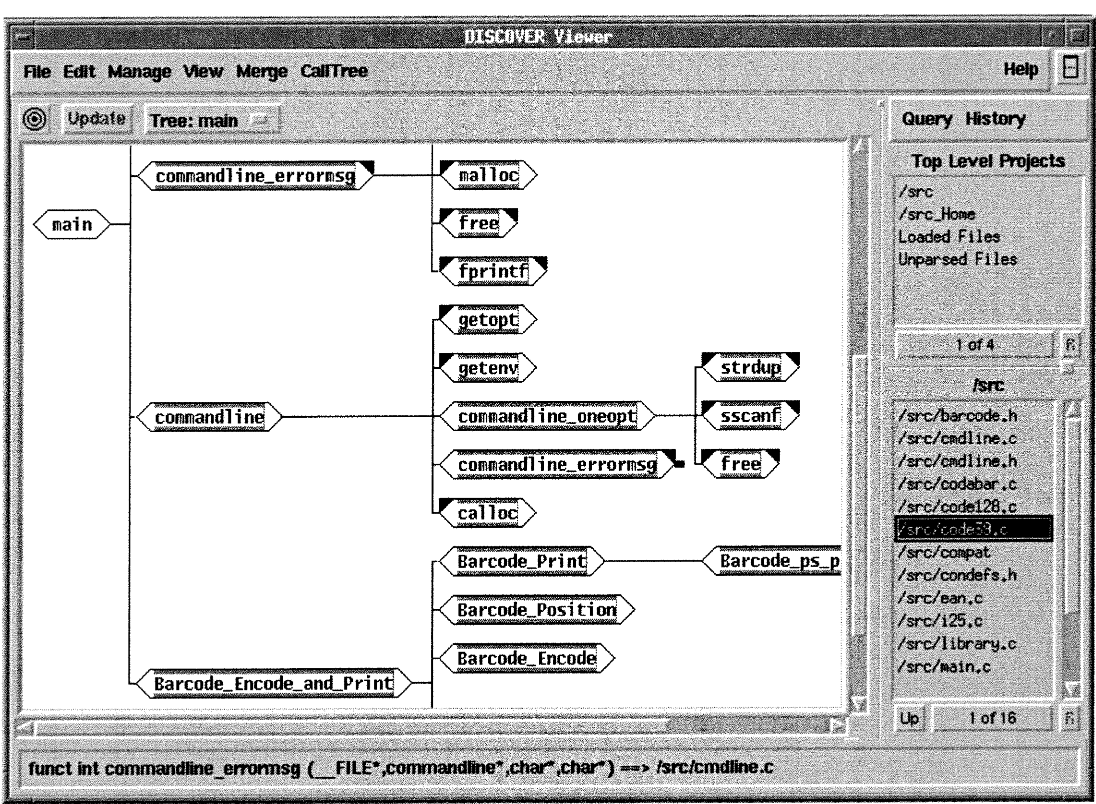


Figure 19 An Example in View of Discover's DeveoperXpress

Users also can extend the functionality available in Developer*Xpress* by adding advanced graphical views to basic browsing. It also provides access to a TCL-based scripting language and Tree Pattern Matching (TPM) technology. With these new analytical technologies, software designers can create additional syntactic patterns of own interest.

Parsing-based tools have their deficiencies, too [2]. Although they are very well suited to the recognition of programming-oriented concepts, they cannot be used in searching for semantic information.

5.2 Extracting Information

As mentioned in Chapter 4, the proposed approach makes use of the information on two relations, *VR-relation* and *RR-relation*. *VR-relation* is the usage of each variable in a set of routines, while *RR-relation* is the call relation among routines, which renders information indicating that what other routines are called or used by each routine.

For the purpose of acquiring the information extracted from a target software system as the input of the approach, we need to use two other tools, Access and Tree Pattern Matching (TPM) in the Discover toolkit. Access is a TCL-based language that allows one to write scripts to perform complex operations. Once written, an Access script may be used again and again: the same kind of analysis applied to one project may be again on another project. Access script may write output to files in a certain format TPM extends Access to provide access to Abstract Syntax Trees (ASTs)-complete parse trees constructed and saved while building the information model. By searching these parse trees for specific constructs, TPM can find many different constructions of interest. Our Access script gets the following information from a source code:

- The title of subject software system;
- The number of global variables in the system;
- The number of functions in the system;
- The name and ID of each variable in the system;
- The name and ID of each function in the system;
- For each variable, a set of functions that use it;
- For each function, a set of other functions that are called by this function.

The following describes how our Access script extracts required information from a subject software system and discusses some aspects of acquiring accurate and complete information.

First, our script extracts such general information, as the title of project, total number of functions and variable defined, in a subject system. The script segment can perform the tasks. In this segment, "home_proj", "defines" are query commands; "cname" is string attribute; "-functions", -"variables" are entities provided by Access. Query commands are allowed to find entities, such as home project, functions and variables in the information model built for a system. String attribute "cname" returns the name of the specified entity.

```
puts $fileId "@project [cname [home_proj]]"
set funs_num[size [defines -functions /]]
set vars_num [size [defines -variables /]]
```

In view of some deficiencies from the global-based method by Liu and Wilde, Livadas and Johnson noticed that first, in programming languages allowing nested procedures (like Pascal), those variables which are visible for several procedures should be considered as global variables; Second, although a global variable is passed as parameter to a function, the receiver function should notice that the

parameter originally is global variable. The algorithm for the object finder described in Liu and Wilde (1990) is unable to find the threads that would be caused by such bindings.

As we can see, our studied systems are all implemented in C programming language that does not allow nested procedures. Thus, those variables that are visible in more than one procedure should be considered as global variables. Furthermore, using `Access` script and `TPM`, we can extract the information indicating which functions take a certain global variable as their actual parameter. So, the *VR-relation* are a set of binary relations that for each global variable indicate not only which routines use it directly but also which routines take it as their actual parameter.

```
set data_set [sort [defines -variables /]]
set nmain main

foreach data $data_set {
   puts -nonewline $fileId "[itag $data]\#[cname $data]\{"
   set datafun [sort [filter funct [where used $data]]]
   set argfuns [filter funct [tree navigate { r( <- ast_expr )
      call_expr } [instances $data]]]
   set ufuns [sort [set_union $argfuns $datafun]]

   set j 0
   foreach sfun $ufuns {
     set singlefunstring [cname $sfun]
     if { [size [where defined $sfun]] != 0
            && [string compare $singlefunstring $nmain] != 0 } {
       if { $j == 0} {
         puts -nonewline $fileId "[itag $sfun]\#[cname $sfun]"
       } else {
         puts -nonewline $fileId " [itag $sfun]\#[cname $sfun]"
       }
       incr j 1
     }
   }
   puts $fileId "\}"
}
puts $fileId "\n"
```

This above segment of Access script is to extract the above-mentioned *VR-relation* and to write its output into a file, called fileId. The statement, "set data_set [sort [defines -variables /]]", assigns a set of all the global variables to data_set. For each variable $data in the set, a set of those functions that directly use it could be found by the statement, "set datafun [sort [filter funct [where used $data]]]", where "[where used $data]" finds those entities that use the variable, and then "[filter funct [entities]]"applies attribute-expression, funct to each member of entities and creates a return set that contains only those functions. For the same variable, a set of those functions that take it as their actual parameter could be acquired from the statement "set argfuns [filter funct [tree navigate { r(<- ast_expr) call_expr } [instances $data]]]", where "[tree navigate { r(<- ast_expr) call_expr } [instances $data]]]" searches its AST for those entities that use all instances of the variable by using the TPM tool , and "[filter funct [entities]]" filters these functions out. It should be pointed out here that the AST is highly structured representation of a body of source code. It encapsulates all of the syntactical properties of the source in a format that can be formally manipulated. Each element or node in the tree represents a syntactical component and may have one or more sub-nodes. "set ufuns [sort [set_union $argfuns $datafun]]" is used to unite those functions that directly use the variable and those functions that the variable as their actual parameter so that we can get a complete set of functions that access the variable.

The appropriate selection of input data and routines strongly affects the results of object identification. Our algorithm turns out to be sensitive to items that possess all features. In a C system, too many routine calls and many global variables are contained within the function "main". In that case, it may be preferable to ignore this

function. As a result, we have removed the routine "main" from a list of routines which access a global variable and those global variables the routine "main" uses uniquely from the C systems. The reason is that a global variable appearing in "main", which is not accessed by any other routine, can be considered as a constant. The condition , "[string compare $singlefunstring $nmain] != 0" in the segment, is set to exclude "main".

Considering that extracted information is in the scope of the whole system and the different variables or different functions with same name occur possibly (for example, different global variables are given the same name from different files in C programming language), each variable and each function are prefixed to an ID. The ID consists of a unique tag number followed by as symbol #, which is assigned to each entity by `Discover` automatically during the internal representation generation step. "[itag $data]\#" and "[itag $sfun]\#" in the segment return a string that uniquely identifies a variable and a function respectively.

Similarly, we can also extract the *RR-relation* using `Accept` script. The following is a part of segment for extracting *RR-relation*. First, the statement "set pfun [sort [defines -funct /]]" finds a set of all defined functions in the subject system and assigns them to "pfun"; Then for each function of them, the statement "set psubfun [sort [uses -functions $fun]]" extract those functions that are used by it. These functions may include not only defined functions but also built-in functions provided by library. However, the later functions are not relevant to the domain of the subject system and should be excluded. We use a constraint "[size [where defined $subfun]] != 0" to prevent these functions from being included.

```
set pfun [sort [defines -funct /]]

foreach fun $pfun {
  set funstring [cname $fun]
  if { [string compare $funstring $nmain] !=0 } {
    puts -nonewline $fileId "[itag $fun]\#[cname $fun]\{"
    set psubfun [sort [ uses -functions $fun ]]
    set psubfun_size [ size $psubfun ]
    set n 0
    foreach subfun $psubfun {
      set subfunstring [cname $subfun]
      if { [size [where defined $subfun]] != 0 && [string compare
$subfunstring $nmain]!= 0 } {......}
......
```

A sample of the input file created by our script is showed in Figure 20.

```
@project Barcode
@routinesNames{3/421#Barcode_128_encode ...}

@dataInfo
3/995#alphabet{3/683#Barcode_pls_encode 3/682#Barcode_pls_verify}
3/490#alphabet{3/524#Barcode_39_encode 3/520#Barcode_39_verify}
3/178#alphabet{3/207#Barcode_cbr_encode 3/201#Barcode_cbr_verify}
...

@routinesInfo
3/421#Barcode_128_encode{3/417#Barcode_128_make_array}
3/405#Barcode_128_verify{}
3/524#Barcode_39_encode{3/521#add_one}
...
```

Figure 20 A sample of Input File for the Approach

5.3 Properties of the Prototype

Given an input file, based on the setting of several parameters by a user or by default, the prototype presents the result of identified objects. The result includes the following important information:

- Setting of parameters for the result which consists of clustering tree height, cohesion threshold value, and each value of three modes' weight;
- Information on the subject system including the system title, the total number of global variables, and the total number of functions in the system;
- Result summary containing minimum cohesion which indicates that minimum value of cohesion for the system can be set, the number of identified objects, the values of average coupling and average cohesion which are used for evaluating overall quality of the result of identified objects, the iteration time;
- Detailed information on the resulting objects where for each identified object, it displays which global variables are contained in the object, which functions refer to these variables.

Figure 21 shows an example of the result displayed in an interface of the prototype.

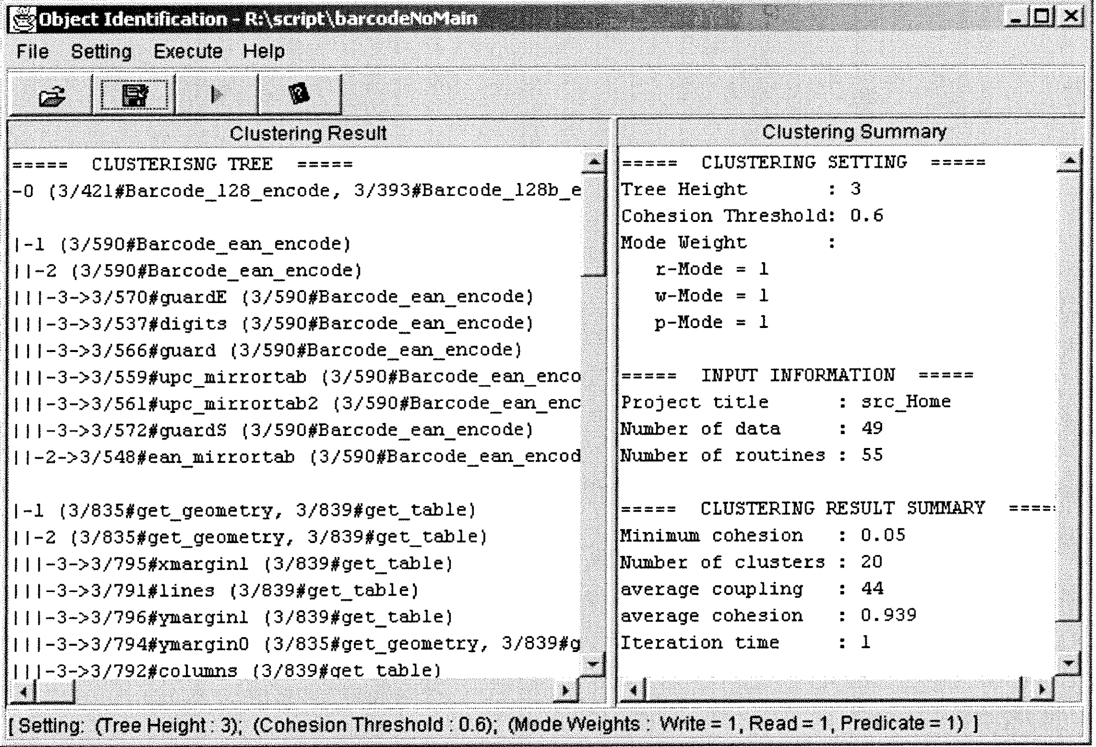


Figure 21 Interface of the prototype

The prototype provides users also with the other base facilities that may support more typical object identification tasks through setting parameters based on the characteristics of the subject system. One user is allowed to set any of the five parameters individually and also the prototype offers defaulted values for the parameters shown in Figure 22.

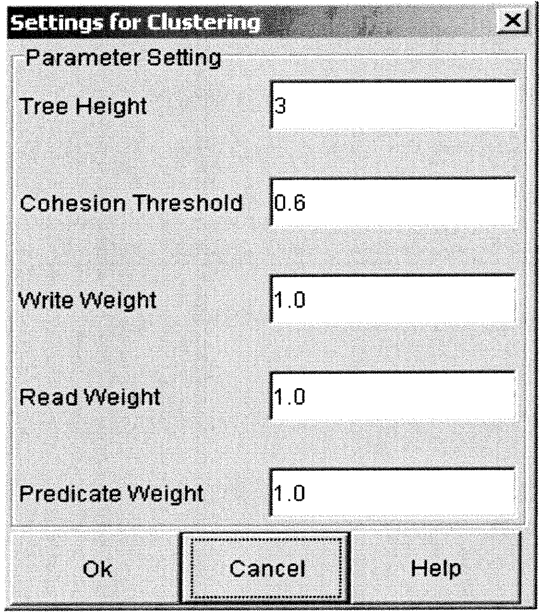


Figure 22 Parameter Setting

Besides an object identification method itself, the quality of object identification is also affected by inherent character of a subject system to a large extent, such as its domain, programming style, size, and the degree to which modules are implemented in the system. The setting of these parameters provided by the prototype enables one to mitigate unfavorable impacts on the object identification due to these factors.

Clustering Tree Height Its value ranges from 2 to infinite and its defaulted value is 3. A hierarchical tree can be grown to arbitrary height. As the tree gets deeper, the cost

of the above phases grows substantially. In general, a relatively low height of the tree is set for those small-size systems or systems implemented with an appropriate degree of modules. However, for those large-size systems or system with low degree of modules, one should increase its height correspondingly, which makes the reordering operator introduced into the proposed approach more powerful for the purpose of enhancing the quality of object identification.

Cohesion Threshold its value is between 0.0 and 1.0. There could be two extreme cases for a given system, one is that all variables are grouped into an object with a minimum cohesion as shown in Figure 21, and the other is that each variable turns out to be an object with cohesion value 1. The setting of Cohesion Threshold provides one with the means of finding out best result of object identification for a given system depending on its inherent character. For example, for a system with large size, low degree of modules, a relatively low value should be assigned to it.

Model Weights their values range from 1.0 to 10. The relationship between a routine and a global variable simply indicates that the routine uses the variable. In our case, the way in which the routine uses the variable is important. We define three modes: modification or write mode (w) when the routine modifies the value of the variable, access or read mode (r) when it accesses its value to compute something else, and predicate mode (p) when the variable is used to control the execution of the routine (in a predicate). In general, one can set a high value for write mode weight and a low value for read or predicate weight. This classification is based on the works of [46] and [41] on module coupling. This improvement can help us for two reasons:

- A global variable in a system that has no link in (m) mode can be considered as a constant, and removed from the graph (such decisions are not easy to make when pointer arithmetic is used).
- When we identify methods, the mode can be considered in conflict situations.

These adjustable parameters offer more flexibility. On the other hand, when the maintainer wants to search for candidate objects similar to those already found, these parameters can be automatically calibrated by the set of known objects.

Chapter 6 *Evaluation of the Algorithm*

In this chapter the proposed approach is applied to the three small real-life systems and the results are compared to those identified by human as well as the results obtained by genetic algorithm, called GOAL [35]. With respect to the comparison, an approximate matching technique is introduced to evaluate the quality of the approach proposed. It shows the application of the approach to these simple programs and discusses the results of a set of case studies, which were carried out in order to assess its accuracy, strength, and effectiveness, while identifying its major limitations.

6.1 Systems Studied

The systems used for evaluating the algorithm are all small size real-life and diverse C systems (see Figure 23 for their characteristics). ***Barcode*** is a business tool designed to convert text strings to barcodes. It supports a wide variety of encoding standards and creates encapsulated/postscript output. ***Jalote*** is the software for scheduling a set of courses offered by the Computer Science Department and ***Sga-c*** is a package for `Goldberg's Simple Genetic Algorithm` and a C-

language translation and extension of the original `Pascal SGA` code presented by Goldberg.

In Figure 17 the systems, their name, their version, the lines of code, the number of files, the number of user-defined types and the number of global variables and routines may give an overview on these systems. All figures about program length in terms of lines of codes in the Figure include comments and blank lines. Most systems have additional libraries that often encapsulate platform dependencies. These libraries were not investigated. Figure 23 lists only the size of the core systems that were analyzed. The number of global variables does not include constants.

System Name	Version	# Files	Lines of Code	# User Types	# Global Variables	# Routines
Barcode	0.96	14	3.8 KLOC	7	56	56
Jalote	1.0	1	1.6 KLOC	4	19	40
Sga-c	2.0	13	1.2 KLOC	4	28	42

Figure 23 Suite of Analyzed C Systems

For the purpose of evaluating the algorithm, although the size of these systems is small and the limited extent of the case studies may not allow definitive conclusions there are a number of considerations that can already be drawn from evaluation results of the diverse systems. The general objective of small-scale evaluation is not to yield a definite empirical proof for the usefulness of a method for all kinds of systems and settings but to learn about the strengths and weaknesses of a method and to investigate where further research should be directed.

These systems are written in C language. The decision to use them has practical reasons and reasons that lie in the language as such. Many legacy systems are written in C and many large C systems are available in the public domain. Furthermore, C is widely used as target language in the reverse engineering community, which allows comparable results. C supports abstraction by allowing the user to define his own types and by offering means to hide details of the implementation. Yet, the support for information hiding is quite limited and commonly unused such that reverse engineering can make a real contribution to program comprehension of C programs. All that makes C an interesting language from the reverse engineering researcher's point of view: There is abstraction in the language, yet not enough; programs are designed with the ideas of information hiding in mind, yet these ideas are often ignored. Last but not least, C is anything else than a toy language: it has many idiosyncrasies, such as pointer arithmetic, an unsafe type system, or gotos that make analyses of C programs difficult. If an approach works for C, it is likely that it also works for languages that are at a higher level of abstraction than C.

6.2 Reference Corpus

In order to establish a comparison point for assessing accuracy, strength, and effectiveness, as well as major limitations of our algorithm, a list of reference objects (short references) decomposed manually from the systems must be available. These reference objects can be used for statistical analyses. For the evaluation, we compared the objects proposed by our approach, called candidate objects (short candidates) to corresponding reference objects. The systems ***Barcode*** and ***Sga-c*** were analyzed and the reference objects were established by two groups of computer science graduates for both of them. Furthermore, they manually developed the object-oriented redesigns for these two systems. At that time, the graduates were taking the course Advanced Software Engineering in Département d'Informatique et de Recherche Opérationnelle, Université de Montréal and fulfilled the work as their projects in the course. All of these graduates acquire solid and deep knowledge about

reverse engineering and object oriented analysis and design. They had at least 4 years of programming experience and were familiar with the programming language C. The system ***Jalote*** is originally written in C and its author also translated it into a functionally equivalent C++ program, so they provide a reasonable basis for comparison.

These students construct class diagrams in UML for the systems ***Barcode*** and ***Sga-c***. Also we reconstruct the class diagram in UML for the system ***Jalote*** based on its C++ program. Figure 24 summarizes what kind of model elements occur in UML class diagram for these respective systems. A ⊥ means that the model element does not appear in the class diagram, and a √ says that the model element exists in the diagram. Here we should mention that for ***Barcode,*** since the students add some classes to their OO model design aiming at offering interface, these classes are not considered as the reference objects in reference corpus in comparison to candidate objects.

System Name	***Barcode***	***Sga-c***	***Jalote***
No. of Classes	24	10	21
Inheritance	√	⊥	√
Aggregation	⊥	⊥	√
Composition	√	√	√
Abstract Class	√	⊥	√
Template Class	⊥	⊥	√
Interface	√	⊥	⊥

Figure 24 Summary of model elements for reference corpus

6.3 Classification of Matches

With respect to the comparison of candidate objects to reference objects, the elements of objects are basically sets of global variables. Its important information is

whether the elements of one object are a subset of the other's elements. For this purpose, we use an approximate matching technique proposed by Girard, Koschke, and Schied. The adoption of the approximate matching is to accommodate the fact that the distribution of routines and global variables into objects is sometimes subjective and, pragmatically, we have to cope with matches of candidates and references that are incomplete, yet "good enough" to be useful. "Good enough" means that candidate and reference overlap to a large extent and few elements are missing.

The approximate matching introduces a **partial subset relationship**, $\subseteq_p$ between two sets A and B. the relationship is defined as follows:

$$A \subseteq_p B \text{ if and only if } \quad \frac{|A \cap B|}{|A|} \geq p \tag{6.1}$$

where the tolerance parameter p in this relationship ranges 0 to 1 and can be specified by the maintainer. If p is set to 1.0, A must be completely contained in B.

Based on the definition of partial subset, we compare two objects with each other to ascertain their degree of overlap. More precisely, we treat one object S as a match of another object T if S is a partial subset of T (denoted by $S \subseteq_p T$). For the results reported in this chapter, $p = 0.7$ is assumed, i.e., at least 70 percent of the elements of S must also be in T. This number is arbitrary, but motivated by the fact that at least three elements of a four-element object must also be in the other object to be an acceptable match.

We use the same approximate matching criteria as those proposed by Girard, etc., setting p as 0.7 to classify the quality of candidate objects into three categories,

Good, **Ok**, and **Bad** according to the degree of matching between corresponding objects. The following is described precisely and in detail.

- **Good** when the match between a candidate object, *CO*, and a reference object, *RO,* is close, (i.e., $CO \subseteq_p RO$ and $RO \subseteq_p CO$ where $p \geq 0.7$). This category requires that a great proportion of elements in corresponding objects be matched and a quick verification in order to identify the few elements which should be removed from or added to the candidate object. This case is denoted as 1~1.
- **Ok** when the relationship holds only one direction for a candidate object, *CO*, and a reference object, *RO*.
  - $CO \subseteq_p RO$ but not $RO \subseteq_p CO$. *CO* is a subset of *RO* to a large extent. The candidate is too detailed. This case is denoted as n~1.
  - $RO \subseteq_p CO$ but not $CO \subseteq_p RO$. *RO* is a subset of *CO* to a large extent. The candidate is too large This case is denoted as 1~n. Partial matches of this type require more attention to split, combine, or refine a candidate and reflect the fact that multiple **Ok** matches may exist for a given *RO* or *CO*.
- **Bad** when the relationship cannot hold in both directions for a candidate object and a reference object. This category indicates that a candidate object is not close enough to the reference object to guide the software engineer's work in object identification.

Example. Consider the example in Figure 25. CO_1 and RO_1 are a good match. Because only partial matches are required, there can be another reference RO_4 (with $R_4 \cap R_1 = \varnothing$) that is a partial subset of CO_1 (of $CO_1 \setminus RO_1$, more precisely). CO_2 is an **Ok** match with RO_2, and so is CO_3. CO_2, CO_3, and RO_2 constitute an n~1 match. That is, the technique has produced finer-grained candidates than what was expected. Note that we cannot necessarily conclude that $CO_2 \cup CO_3$ and RO_2 are a good match because RO_2 could be much bigger than $CO_2 \cup CO_3$. RO_3 and CO_4 constitute a 1~n

match, where no other reference than RO_3 can be matched with CO_4. CO_5 and RO_5 do not match at all.

As the example indicates, it is not enough just to count the matches in order to judge the detection quality of a technique. For example, RO_3 is a partial subset of CO_4 and, therefore, considered at least an Ok match. However, CO_4 could be huge and the match could just be coincidence. The next section proposes a measurement for detection quality based on multiple aspects that considers this imprecision.

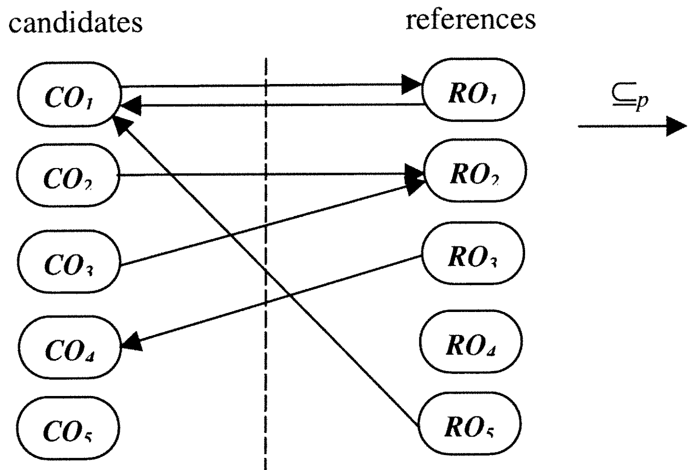


Figure 25 Example correspondences of candidates and references

6.4 Accuracy and Recall Rate

Since the partial subset relationship is used to establish a match, the matching candidates and references need not be equal. That is, there may be elements of the candidate not in the reference and vice versa: $CO \setminus RO \neq \emptyset$ and $RO \setminus CO \neq \emptyset$. In other words, there may be a flaw in a good match; even more so for **Ok** matches because of (let RO be a reference and CO_i be candidates for which $CO_i \subseteq_p RO$ holds):

$$\bigcup_i CO_i \subseteq_p RO \not\Rightarrow RO \subseteq_p \bigcup_i CO_i$$

In the presence of these imprecise matches, in order to indicate the quality of imperfect matches of candidate and reference elements, an accuracy factor has been associated with each match that measures the quality of a relation between a candidate and a reference. The **accuracy** between a candidate and a reference is computed as follows:

$$accuracy(C,R) = \left|\frac{C \cap R}{C \cup R}\right|$$ Where C and R stand for a set of entities.

For matches between more than two objects (1~n and n~1), the union of all n objects is used for the accuracy. Note that the accuracy of good matches can also be below the threshold of the partial subset relationship. For example, if RO and CO both have 10 elements and 7 elements of RO are in CO and 7 elements of CO are in RO, then $RO \subseteq_p CO \wedge CO \subseteq_p RO$ holds and, hence, RO and CO are a good match. However, the overlap of RO and CO is only $7/13 = 0.54 < p = 0.7$.

If there are some candidate objects with no sufficient relation to the reference objects, they are called **false positives**. These are candidates which are not represented in the reference and, thus, are false candidates. If there are some reference objects with no sufficient relation to the candidate objects, they are called **true negatives**. The existence of true negatives leads to unmatched references. The best evaluation result would contain no false positives and no true negatives. The number of false positives and the number of true negatives are two important aspects in a comparison of a set of candidates with a set of references to consider when matches have been established. The followings describe them more precisely.

Number of false positives: The number of candidates that neither match a reference nor are matched by any reference, i.e., candidates that cannot be associated with any

reference. Technically speaking, these are candidates that are neither involved in a 1~1, 1~n, nor n~1 match. This number should be 0.

Number of true negatives: The number of references that neither match a candidate nor are matched by any candidate, i.e., references that are not even partially detected. Technically speaking, these are references that are neither involved in a 1~1, 1~n, nor n~1 match. This number should be 0.

To quantify the overall detection quality of the algorithm, the **recall rate** is computed. The recall rate abstracts from the level of granularity – since all positive relations are treated equally – and ignores the number of false positives. Thus, the false positives have to be given extra inspection. The recall rate is computed as follows:

$$\text{recall rate} = \frac{\sum_{c \in Good} accuracy\ (C) + \sum_{c \in Ok} accuracy\ (C)}{|Good| + |Ok| + |\text{true negatives}|} \tag{7.2}$$

In the equation, $|x|$ denotes the number of elements in the set x. $|Good|$ denotes the number of 1~1 relations and $|Ok|$ denotes the number of both 1~n and n~1 relations. In the following section, the evaluation of the algorithms is discussed.

Different aspects are considered to allow a detailed survey of the results. The recall rate and the number of false positives would suffice to give an overview of the quality of the technique. However, a simple example shows that those two numbers alone might mislead the reader: If a technique proposes one single candidate that is actually the union of all references, the recall rate is 100% and there are no false positives. Nevertheless, a better result would match each reference separately.

6.5 Case Studies

This section compares the objects identified by our approach with those identified manually, presents, and discusses the results of their comparison. It is pointed out that in the case studies, we set the same value 1 for the mode.

6.5.1 Case Study 1

For *Barcode*, an independent, manually developed, object-oriented redesign made by a group of students, exists The number of identified objects is 20 in total and the iteration time is 1. Comparing these candidates with those classes in the redesign for this system using the above mentioned matching criteria, we obtained 6 *Good*, 7 *Ok* objects, 5 false positives, and 6 true negatives. Its recall rate is 0.641.

The results do not yield the complete redesign, but they constitute the core classes of the independent redesign. One difference is that no identified objects can match some classes related to interface which is added to the redesign. A second difference is that in the redesign some OO model elements, such as inherence, composition, are used which cannot be considered in our approach. The third difference is that in the redesign domain knowledge is used to further refine certain classes (for example, a separate "HelpMessage" class is included). However, this separation is not explicitly present in the legacy system. For this reason, it is not included in those identified objects. To some extent, these differences lead to the increase of the number of both false positives and true negatives.

By looking more closely at source code of *Barcode*, we find out that the programmers of *Barcode* followed the information hiding principle in its implementation and restricted the access and modification variables to a limited number of routines, so that the modularity of the system is enhanced. This might have contributed to a number of exact matches between candidates and references.

6.5.2 Case Study 2

Sag-c has a smaller size compared to *Barcode.* The graduates who made the redesign of the system are acquainted with genetic algorithm and domain knowledge of the system. As a result, the redesign provides a complete list of references without losing details. With the settings of tree height 7 and cohesion threshold 0.65, the approach has its best results with the total number of 12 objects in iteration time 3. Based on matching criteria, we have 6 *Good*, 5 *Ok* objects and 1 false positive and 3 true negatives. Its recall rate is 0.75.

For this system, high recall rate, low numbers of both false positives and true negatives are given. We can expect this result because of the following two contributing factors. The first factor is that the programmers of the subject system take advantage of the means of the programming language C for information hiding of global variables. These variables are mostly declared static instead of declared in header files (they can only be declared there as external). The second factor is that there are a relatively high number of references that are detected as a 1~1 match by the approach since the most of the same domain-related data in the system are used both by the approach and the redesign of the system.

6.5.3 Case Study 3

Jalote is redesigned and implemented functionally equivalent to its original program in C++ by the same author. Setting tree height and cohesion threshold as 5 and 0.53 respectively. The approach identifies 14 objects in total in iteration time 4, among which are 3 Good, 7 Ok objects and 3 false positives. Its recall rate is 0.53 and the number of true negatives is 7.

In the case of *Jalote*, the approach has a relatively high number of both false positives and true negatives in spite of its smaller size. A more detailed investigation

on this reveals that there are many user defined types such as *struct* in C program are simply replaced with classes in C++ while those variables with these types are mostly declared in the form of an array in the C program, which leads to more true negatives. Furthermore, some variables grouped in an identified object no longer appear in its corresponding class in C++ since values of them can be obtained by calling some member function in the class. For example, the value of variable `RoomSize` in C is available by calling a member function `tellSize` in C++, which may increase the number of false positives.

6.5.4 Summary of Evaluation Results

The following figure summarizes the evaluation results of the approach applied to the three systems.

System Name	Parameter setting		Evaluating Results				
	Cohesion Threshold	Tree Height	Recall Rate	#Good	#Ok	#False Positives	#True Negatives
Barcode	0.55	5	0.641	6	9	5	6
Jalote	0.53	5	0.529	3	7	4	7
Sag-c	0.65	7	0.75	6	5	1	3

Figure 26 Summary of evaluation results

According to these summaries, the effectiveness of the approach strongly depends upon its subjective system. When those figures of the number of Good, Ok, false positives and true negatives in Figure 26 are represented in percentage of the total identified objects as shown in Figure 27, the difference is observed obviously.

Finally, in order to evaluate the overall quality of the approach to the three systems, we calculate for each of the three categories of identified objects (*Good, Ok, Bad*) the average of the three percentages corresponding to the three systems (see figure 28). We note the relatively high percentage of *Ok* of the three categories of objects, 46%. This indicates that many of identified objects by the approach need more attention to split, combined, or refined

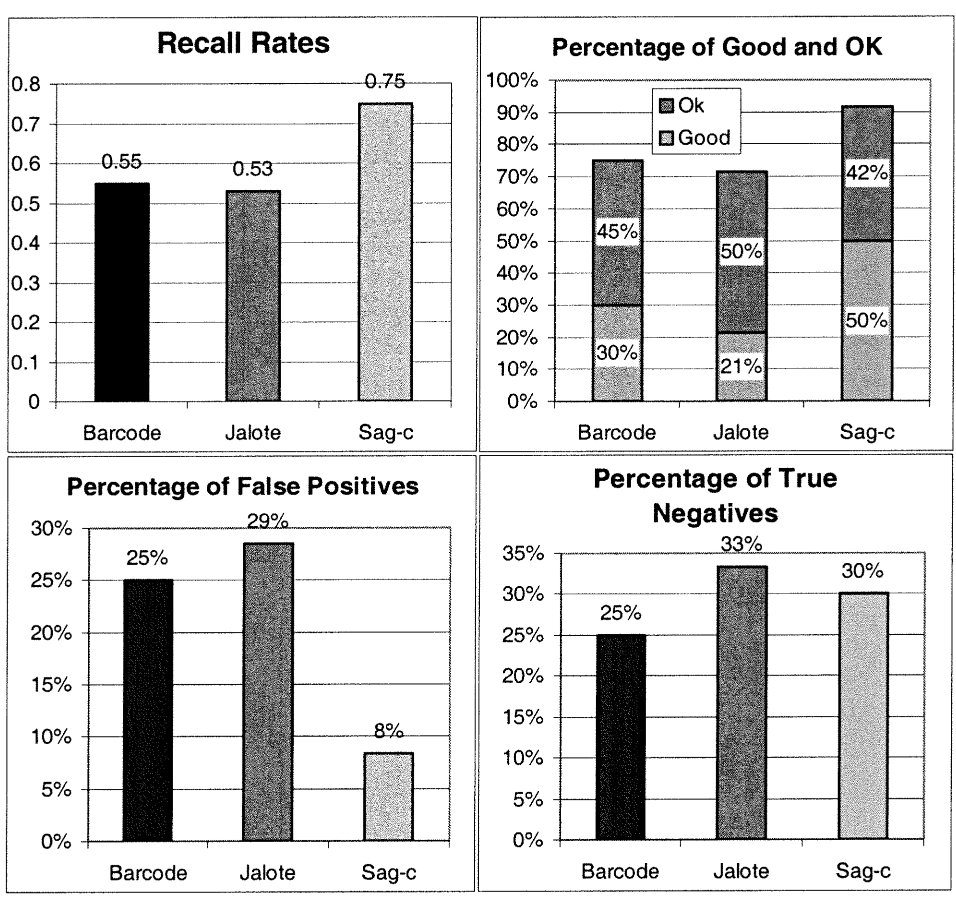


Figure 27 Evaluation results in percentage

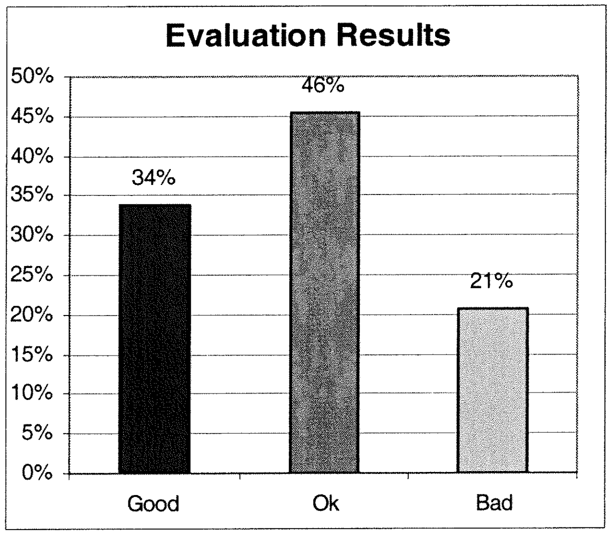


Figure 28 Overall quality of the approach

Furthermore, the results of the comparison of the identified objects by conceptual clustering based approach (CCBA) to the identified objects by GOAL for each system are given in 29.

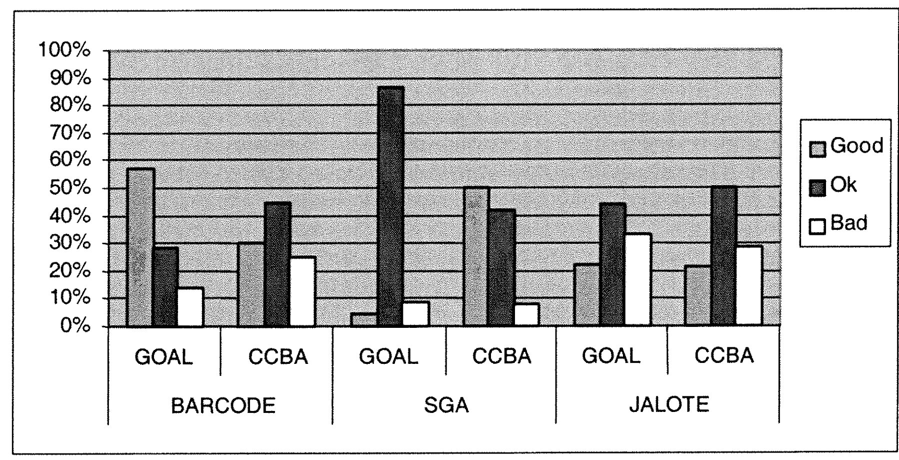


Figure 29 Summary of CCBA and GOAL evaluation results

The two algorithms show almost the same percentage of false positives except for Barcode where GOAL performs better. The performance of the two algorithms

depends on the systems. The results of SGA are better than those of Barcode, which in turn are better than those of *Jalote*. CCBA performs better than GOAL on two systems. This can be explained by the fact that GOAL has more parameters to set than CCBA.

Another interesting issue is the sensitivity of the approach to the COHESION parameter. As shown in Figure 30, the results of the application of the approach on SAG vary significantly from one COHESION value to another.

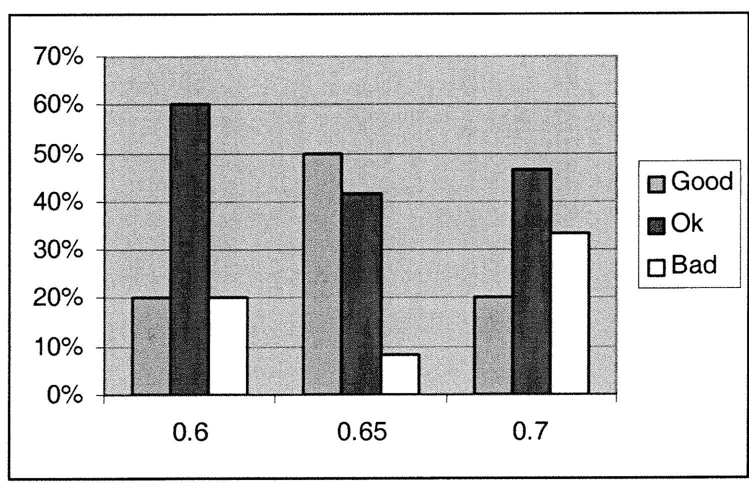


Figure 30 Sensitivity of the approach to COHESION variations

6.6 Discussions and Lessons Learned

Even if we didn't test our algorithms on a large set of system, we can reasonably claim that they perform well when the migrated system is of good quality. However, if the system is poorly written, the algorithms cannot produce miracles.

Although overall results of the evaluation of the approach are encouraging in view of the limitation of automated object identification techniques, the phenomenon that the number of false positives is relatively high deserves special attention, and hence we make their analysis.

The approach proposed some objects for which no corresponding reference objects (classes in its redesign) existed and, therefore, were classified as false positives. We investigated these false positives to learn more about the weaknesses of the approach. It turned out that a few false positives are indeed correct positives; some of these were too small to be considered by the students as objects, others were simply overlooked by them. The analysis of false positives revealed certain common patterns that were generally found in the set of false positives proposed by the approach.

Some global variables are only referenced by one routine; thus, they act as static local variables of this routine but the programmer did not take advantage of the ability in C to express this explicitly. A routine with such static local variables alone can hardly be considered an object in a narrower sense -- even though the local variables indeed clearly belong to the routine

Variables used at many places in the system often represent global system parameters, e.g., variables that indicate whether a certain command line switch was set when the program was invoked. Often, it is recommended to exclude frequently used variables [57]. However, simply excluding frequently used variables may also affect variables of an abstract data object that the programmer made public. A more reliable method is to exclude variables that are directly data dependent on the parameter *argv* of the main routine that contains the command line arguments of the invoked program in batch-oriented systems. It is still not clear how to distinguish these from frequently used public variables of an abstract data object.

The main lesson we learned from this problem was that we need human intervention to decide which data are domain related. Our approach is not able to know automatically which components under analysis belong to a domain-independent

library, and which components belong to the application domain. We consider, however, that this kind of information can be easily obtained from the maintainers. Once we know which routines should not be analyzed, our approach is able to work properly without further help from the maintainer.

Furthermore, due to the degree of vagueness of reasonable decompositions and the complex semantic issues involved, the user should be integrated into atomic object identification. For this reason, our approach allows the user to set a key parameter, cohesion threshold, which provides the effective ways of user integration. The effectiveness of the approach depends upon system characteristics, like degree of information hiding, proper module decomposition. If programmers followed the information hiding principle, the approach would detect all abstract data objects without any false positives. However, they hardly exist in this case.

Chapter 7 *Conclusion*

This chapter summarizes the work of this thesis and proposes further research directions that seem to be worthy to be explored.

7.1 Thesis Summary

Object identification is the key activity in migration of a legacy system to an object-oriented one. It helps to understand system design and facilitate the reuse of existing components contained in the system.

In this thesis, we presented a new approach aimed at identifying objects in procedural code more effectively. It differs from other work by the fact that it borrows part of its inspiration from the artificial intelligence sub-field of conceptual clustering and focuses on the identification of objects, which should be internally cohesive; Meanwhile inter-dependency between them should be kept as loose as possible. The prototype we built can work in an automatic fashion. It is also open to

human intervention when an expert is available. These adjustable parameters offer more flexibility in searching for candidate objects similar to those already found.

Realizing the inherent advantageous characteristics of Cobweb system, unsupervised, incremental, hierarchical, and base-level effect, as stated in Chapter 3 we first proposed the approach to object identification which inspires Cobweb. Its performance turns out to be effective and encouraging. One of its advantages is that it yields a dendrogram of clustered entities instead of a set of flat candidate objects. Significant experience with the use of conceptual clustering for object identification is acquired.

Of utmost importance with high quality of identified objects is the appropriate use of information extracted from legacy code. We identify two kinds of structural information to be used by the approach: one is the usage of each variable in a set of routines; the other is the call relation among routines. The two relations are supposed to significantly capture the basic structural information leveraged for object identification.

Aimed at enhancing the quality of object identification, some works we have done are listed below

- We introduce a process of iterative optimization into the approach to seek higher quality of objects decomposed from a legacy system. The process compares the qualities of tow sequential clustering trees according to their total sum of couplings between candidate objects and terminates when there is no further improvement in clustering quality.

- We propose a new clustering algorithm, called *OI-Cobweb* that makes some modifications on Cobweb. First, *OI-Cobweb* creates new clustering criterion functions (objective functions). The criterion functions are directly associated to

the well-known object oriented design metrics (cohesion and coupling), thus the criteria of identifying objects also reflect object-oriented design to a large extent; second, *OI-Cobweb* uses the multi-criteria decision-making to prevent a candidate object from turning out to be a highly cohesive cluster while highly coupling with other objects or a low cohesive cluster while coupling with other objects. This kind of bad objects may be produced while using single criterion decision-making like CU in Cobweb. A desirable characteristic of the multi-criteria making is that first of all it insures every identified objects with a relatively high degree of cohesion and then keeps its coupling with others as loose as possible.

- In order to overcome order effects, we put a *Reordering* procedure into the iterative optimization process and extract the so-called interleaved ordering in each iteration. The introduction of reordering procedure to our approach mitigates ordering effects globally and uncovers better clustering effectively.

The approach has been applied to three small real-life systems and the results are compared to those identified manually. The validation results of the approach are considered to be good concerning recall rate. The further analysis on the results reveals its drawback due to the inherent limitation of automated techniques.

7.2 Future Work

One obvious direction to extend this research is to apply the approach to type-based object identification, which clusters a routine with a set of types of all its formal parameters and the type of its return value. It might be useful to cluster a routine with the type of the global and static variables that it accesses as well as the types of its parameters and return value.

Further study is needed to develop a technique that enables its users to determine the value of cohesion threshold for a given legacy system based on extracted information before they carry out object identification from the system. This work is quite important because the effectiveness of the approach strongly depends upon its subjective system.

The proposed approach uses the relationship between a routine and a global variable. It simply indicates that the routine uses the variable. Sahraoui defined three modes write mode, read mode, and predicate mode, for different ways in which the routine uses the variable [50]. In this thesis, the approach did not take account of the aspect. However, in object identification, a routine which modifies a variable is generally more important than another routine which reads the variable. Hence the former should be given more weight. In future study, the weight factors for different modes should be introduced into the approach to improve its precision. The weights for these modes could be calibrated on results from our case studies by systematic hand tuning.

Bibliography

[1] Basili, V. R., Viewing maintenance as reuse-oriented software development, IEEE *Software*, 7 (1), 1990, 19-25.

[2] Biggerstaff, T. J., Mitbander, B. G., Webster, D. E. (1994), Program understanding and the concept assignment problem, *Communications of the ACM*, 37(5), 72-83.

[3] Biswas, G., Weinberg, J.B., and Li, C. (1994). ITERATE: A Conceptual Clustering Method for Knowledge Discovery in Databases. *Innovative Applications of Artificial Intelligence in the Oil and Gas Industry*. B. Braunschweig and R. Day, Editor. Editions Technip.

[4] Briand, L., Daly, J., Wüst, J. (1998). A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering Journal*, 3(1), 65-117. Also available as Technical Report ISERN-96-14.

[5] Canfora, G., Cimitile, A., Munro, M., Taylor, C. J., Extracting abstract data types from C programs: a case study, *Proceedings of the Conference on Software Maintenance*, Montreal, Canada, 1993, 200-209.

[6] Canfora, G., Cimitile, A., Visaggio, G., Assessing modularization and code scavenging techniques, *Journal of Software Maintenance: Research and Practice*, 7, 1995, 317-331.

[7] Canfora, G., Cimitile, A., and Munro, M., An improved algorithm for identifying objects in code, *Journal of Software Practice and Experienc*e 26(1), 1996, 25–48.

[8] Canfora, G., Cimitile, A., De Lucia, A., and Di Lucca, G.A. (1999), A case study of applying an eclectic approach to identify objects in Code, *Workshop on Program Comprehensio*n Pittsburgh, IEEE Computer Society Press, 136-143.

[9] Chidamber, S.R., & Kemerer, C.F. (1994) Towards a Metrics Suite for Object Oriented design. IEEE Transactions on Software Engineering, 20(6), 476-493.

[10] Chikofsky, E. J., Cross II, J. H., Reverse engineering and design recovery: a taxonomy, IEEE *Software* 7 (1) 1990, 13-17.

[11] Coad, P., Yourdon, E., *Object Oriented Analysis*, Prentic-Hall, 1991.

[12] Coad, P., Object-oriented patterns, Communications of the ACM 35 (9) 1992, 152-159.

[13] van Deursen, A., Kuipers, T., Finding classes in legacy code using cluster analysis, *Proceedings of the ESEC/FSE'97 Workshop on Object-Oriented Reengineering*, Technical Report TUV-1841-97-10, Technical University of Vienna, August 1997. (Internet : `http://www.cwi.nl/~arie/papers/`)

[14] van Deursen, A., Kuipers, T., Identifying objects using cluster and concept analysis, *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, California, May 1999, 246-455.
(Internet : `http://www.cwi.nl/~arie/papers/`)

[15] Dunn, M. F., Knight, J. C., Automating the detection of reusable parts in existing software, *Proceedings of the 15th International Conference on Software Engineering (ICSE'93),* Baltimore, Maryland, 1993, 381-390.

[16] Fisher, H. D. (1987). Knowledge Acquisition via Incremental Conceptual Clustering. Machine Learning, Vol. 2, pp. 139-172.

[17] Fjeldstadt, R.K., and Hamlen, W.T. (1984), 'Application Program Maintenance Study: Report to Our Respondents', Proc. GUIDE 48, IEEE Computer Society Press, April.

[18] Gall, H., Klösch, R., Capsule oriented reverse engineering for software reuse, *Proceedings of the 4th European Software Engineering Conderence (ESEC'93)*, Garmisch-Partenkirchen, Germany, September 1993, *Lecture Notes in Computer Science* 717, Springer, 1993, 418-433.

[19] Gall, H., Klösch, R., Program transformation to enhance the reuse potential of procedural software, *Proceedings of the ACM Symposium on Applied Computing (SAC'94)*, Phoenix, Arizona, March 1994, 99-104.
(Internet: http://www.infosys.tuwien.ac.at/Staff/hg/sac94.ps)

[20] Gall, H., Klösch, R., Managing uncertainty in an object recovery process, *Proceeding of the 5th Inernational Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU'94)*, Paris, France, Julu 1994, 1229-1235.
(Internet: http://www.infosys.tuwien.ac.at/Staff/hg/ipmu94.ps
)

[21] Gall, H., Klösch, R., Kofler, E., Würfl, L., Balancing in reverse engineering and in object-oriented systems engineering to improve reusability and maintainability, *Proceedings of the 18th IEEE Computer Software and Application Conference (COMPSAC'94)*, Taipei, Taiwan, November 1994, 35-42.
(Internet: http://www.infosys.tuwien.ac.at/Staff/hg/compsac-cr.ps)

[22] Gall, H., Klösch, R., Mittermeir, R. T., Architectural transformation of legacy systems, *Proceedings of the ICSE-17 Workshop on Program Transformation for Software Evolution*, Seattle, Washington, April 1995.
(Internet: http://www.infosys.tuwien.ac.at/Staff/hg/icse-ws.ps)

[23] Gall, H., Klösch, R., Mittermeir, R. T., Object-oriented re-architecturing, *Proceedings of the 5th European Software Engineering Conference (ESEC'95)*, Sitges, Spain, September 1995, *Lectue Notes in Computer Science* 989, Springer, 1995, 499-519.
(Internet: http://www.infosys.tuwien.ac.at/Staff/hg/esec-cr.ps)

[24] Gall, H., Klösch, R., Mittermeir, R. T., Application patterns in re-engineering : identifying and using reusable concepts, *Proceedings of the 6th International Conference on Information Processing and Managememt of Uncertainty in Knowledge-Based Systems (IPMU'96)*, Granada, Spain, July, 1099-1106.

(Internet: http://www.infosys.tuwien.ac.at/Staff/hg/ipmu-cr-web.ps)

[25] Gall, H., Weidl, J., Reswolving uncertainties in object-oriented re-architecting of procedural code, *Proceedings of the 7th Intrnational Conference on Inflormation Processing and Managememt of Uncertainty in KnowledgBased Systems (IPMU'98)*, paris, France, Julu, 1998.
(Internet:http://www.infosys.tuwien.ac.at/Staff/hg/publications.html)

[26] Gall, H., Weidl, J., Building object models to source code : an apporach to object-oriented re-architecting, *Proceedings of the 22nd Computer Software and Applications Conference (COMP_SAC'98)*, Vienna, Austria, August 1998. (Internet: link from http://www.infosys.tuwien.ac.at/Staff/hg/publications.html)

[27] Gennari, J. H., Langley, P., and Fisher, D. (1989). Models of incremental concept formation. Artificial Intelligence, 40(1-3): 11-62.

[28] Gluck, M. A., & Corter, J. E. (1985). Information, uncertainty, and the utility of categories. Proceedings of the Seventh Annual Conference of the Cognitive Science Society, pp. 283-287, Irvine, CA: Lawrence Erlbaum.

[29] Girard, J. F., Koschke, R., and Schied, G., A metric-based approach to detect abstract data types and abstract state encapsulation, *Conference on Automated Software Engineering*, Lake Tahoe, *IEEE Computer Society Press,* 1997, 82-89.

[30] Girard, J.F., Koschke, R., and Schied, G. (1999), A Metric-based Approach to Detect Abstract Data Types and Abstract State Encapsulation, *Journal on Automated Software Engineering*, no. 6, October, 357-386, Kluwer Academic Publishers.

[31] Griswold, B., Automated Assistance for Program Re-structuring, *ACM Transactions on Software Engineering and Methodology* 2, 3 (July 1993), pp 228-269.

[32] Horwitz, S., Reps, T., Binkley, D., Interprocedural slicing using dependence graphs, *ACM Transactions on Programming Languages and Systems* 12 (1) 1990, 26-60.

[33] Jacobson, I., Lindström, F., Re-engineering of old system to an object-oreinted architecture, *Proceedings of the ACM AIGPLAN Conference on Object-Oriented*

Programming, Systems, Languages and Applications (OOP-SLA'91), Phoenix, Arizona, October 1991, 340-350.

[34] Klöshch, R. R., Reverse engineering : why and how to reverse engineer software, *Proceedings of the California Software Sumposium (CSS'96)*, Los Angeles, California, April 1996, 92-99.
(Internet : `http://www.infosys.tuwien.ac.at/Staff/hg/css-cr-web.ps`)

[35] Konkobo, I., Master thesis, Identification des objets dans les applications léguées basée sur les algorithms génétiques, Université de Montréal, December 2001.

[36] Koschke, R. (2000), Atomic Architectural Component Recovery for Program Understanding and Evolution, PhD thesis, Institute for Computer Science, University of Stuttgart.

[37] Lehman, M.M., Belady, L. (1985), 'Program Evolution', Processes of Software Change, Academic Press, London.

[38] Lindig, C. and Snelting, G., Assessing modular structure of legacy code based on mathematical concept analysis, *Proceedings of the International Conference on Software Engineering*, Boston, 1997, 349-359.

[39] Liu, S. S., Wilde, N., Identifying objects in a conventional procedural language: an example of data design recovery, *Proceedings of the Conference on Software Maintenance*, San Diego, California, November 1990, 266-271.

[40] Livadas, P.E. and Johnson, T. (1994), 'A New Approach to Finding Objects in Programs', *Journal of Software Maintenance: Research and Practice*, no. 6, 249-260.

[41] Lounis, H., and Melo, W., 1997, Identifying and measuring coupling in modular systems, *8th International Conference on Software Technology ICST'97.*

[42] McKusick, K. M. and Langley, P. (1991). Constraints on tree structure in concept formation, *Proc. of 12th International Joint Conference on Artificial Intelligence*, Sydney, Australia, August, 1 810-816.

[43] Michalski, R. S. (1980). Knowledge Acquisition Through Conceptual Clustering: A Theoretical Framework and an algorithm for Partitioning Data into Conjunctive Concepts. Policy Analysis and Information Systems, Vol. 4, No. 3, pp. 219-244.

[44] Michalski, R. S. & Stepp, R. (1983). Learning from Observation: Conceptual Clusterings. In R.S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), Machine Learning: An artificial Intelligence Approach. Volume I, San Mateo, CA: morgan Kaufman.

[45] Newcomb, P., Reengineering procedural into object-oriented systems, Proceedings of the 2nd Working Conference on Reverse Engineering, Toronto, Canada, July 1995, 237-249.

[46] Offutt, J., Harrold, M. J., and Kolte, P., 1993. A software metric system for module couping, *The Journal of Systems and Software*, 20 (3), 295-308.

[47] Ogando, R. M., Yau, S. S., Liu, S. S., Wilde, N., An object finder for program structure understanding in software maintenance, *Journal of Software Maintenance: Research and Practice*, 6, 1994, 261-283.

[48] Philip, T., Famsundar, R., A reengineering framework for small scale software, *Software Engineering Notes* 20 (5) 1995, 51-55.

[49] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., *Object-oriented Modeling and Design*, Prentice-Hall, 1991.

[50] Sahraoui, H., et al., (1999). A concept formation based approach to object identification in procedural code. *Automated Software Engineering,* 6, 387-410.

[51] Schwanke, R.W., An intelligent tool for re-engineering software modularity, *International Conference on Software Engineering*, May 1991, 83–92.

[52] Fast programming manual. SEMA Group, France, 1997.

[53] Siff, M. and Reps, T., Identifying modules via concept analysis, *Proceedings of International Conferenceon Software Maintenance*, *IEEE Computer Society*, Bari, October 1997, 170-179.

[54] Sneed, H. M. (1996), Encapsulating legacy software for use in client/server systems, *Proceedings of the Third IEEE Working Conference on Reverse Engineering*, Monterey, CA, IEEE Computer Soc. Press, Silver Sand, MD, pp. 104-119

[55] Tenma, T., Tsubotani, H., Tanaka, M., Ichikawa, T., A system for generating language-oriented editors, *IEEE Transactions on Software Engineering* 14 (8) 1988, 1098-1109.

[56] Venkatesh, G. A., The semantic approach to program slicing, *Proceedings of the ACM Conference on Programming Language Design and Implementation (SIGPLAN'91)*, Toronto, Canada, June 1991, *Sigplan Notices* 26 (6) 1991, 107-119.

[57] Yeh, A.S., Harris, D., and Reubenstein, H., Recovering abstract data types and object instances from a conventional procedural language, *Second Working Conference on Reverse Engineering*, July 1995, *IEEE Computer Society Press*, 1995, 227–236.

[58] Zemmer, J.A., 1990, Restructuring for style. *Software Practice and Experience,* 20 (4), 365-389.