Comprehension-Driven Software Clustering

by

Vassilios Tzerpos

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

©Copyright by Vassilios Tzerpos 2001

Abstract

Comprehension-Driven Software Clustering

Doctor of Philosophy

2001

Vassilios Tzerpos

Graduate Department of Computer Science

University of Toronto

A common problem that the software industry has to face is that of maintaining and upgrading legacy software. The fact that the structure of most legacy software systems is not understood makes this a difficult task. However, it is essential that this knowledge be recovered in order to migrate a software system to a new hardware platform or programming language, or simply to enhance its functionality. The proliferation of reverse engineering projects that attempt to regain this knowledge suggests that this is an issue of great importance.

One of the techniques that has been used to deal with the problems that arise from the sheer size and complexity of large software systems is software clustering. Decomposing a software system into smaller, more manageable subsystems can aid the process of understanding it significantly.

In this thesis, we concentrate on a number of open questions related to software clustering, such as methods for evaluating the output of different software clustering approaches, study of the stability of various algorithms, and incremental clustering. We also present an automatic clustering algorithm that is designed to produce decompositions that help our understanding of the target system. This algorithm identifies clusters based on known patterns of decomposition that are observed in large software systems. It also

produces clusters of bounded size and names clusters in an intuitive manner.

The approaches presented in this thesis have been applied to real industrial software systems. The results we obtained demonstrate the effectiveness and usefulness of our techniques.

Acknowledgments

First of all, I would like to thank my supervisor Professor Richard C. Holt for his inspiring interest in my studies. Ric promptly read many versions of this work and made valuable suggestions that improved it significantly. The educational value of working with him cannot be overstated. I feel extremely fortunate to have had him as my supervisor.

I would also like to thank all members of my departmental and final examination committees, Profs. Derek Corneil, Ric Hehner, Alberto Mendelzon, John Mylopoulos, David Penny, and my external appraiser Professor Timothy C. Lethbridge. Their input has benefited this thesis in many ways.

Several people have made the years that I spent at the University of Toronto special. I would like to thank Dimitra and Minas for being a tremendous help at the beginning, Dimitra and Spiros for being great friends throughout, Evan for helping me move numerous times, Petros and Michalis for a lot of good advice and many hilarious moments, Nick for making me go to the gym more often, Theo for an excellent colobaration, Panayiotis for being the captain, Periklis for tolerating my music taste, and Ivan for not getting mad when I preempt aggressively vulnerable vs. not. All my friends in Canada and Greece (and there's too many of them) have contributed greatly to my having a great time during this degree. Sincere thanx to all.

Also, I would like to thank CSER, CITO, ITRC, and IBM Canada, as well as the University of Toronto for providing financial support for this work. A big "thank you" goes to Kathy Yen and Linda Chow for being very helpful.

Many thanks to the people that helped me with this research, in particular Ian McIntosh and the members of the TOBEY group, Spiros Mancoridis and his Bunch team, and Nicolas Anquetil.

The work in this thesis would not have been possible without the support of my family in Greece. The love and kindness of my father Ioannis, my mother Elpiniki, my sister Kalliopi, and my brother Konstantinos, kept me going throughout the years.

My new family in Canada deserves a great deal of credit for the successful completion of this work. Melanie has always been there for me, through good and bad times. She has been the best partner one could hope for. Nutmeg and Kounelitsa have made our lives extra special. I am deeply thankful for the privilege of spending many wonderful years as a member of this family and am looking forward to many more.

Finally, this thesis is dedicated to the memory of my grandma Aspasia and my grandpa Vassilios who passed away during its course.

Contents

1 Introduction **1**

1.1 The software clustering problem 2

1.2 Open issues on software clustering 7

1.3 Research contributions 8

1.3.1 A distance metric for software clusterings 9

1.3.2 Stability of software clustering algorithms 10

1.3.3 Incremental software clustering 11

1.3.4 Clustering for comprehension 11

1.4 Dissertation outline 12

2 Background **15**

2.1 Introduction 15

2.2 Previous Work on Software Clustering 18

2.2.1 Knowledge-based vs. Structure-based approaches 18

2.2.2 Key publications 19

2.2.3 Recent work 21

2.2.4 Observations on the software clustering literature 23

2.3 Cluster Analysis Techniques . . . 24
2.3.1 Similarity Measures . . . 24
2.3.2 Algorithms . . . 26
2.3.3 Observations . . . 30
2.4 Research Challenges . . . 34
2.5 Conclusion . . . 35

3 Comparison of clustering techniques **37**
3.1 Motivation . . . 37
3.2 Flat vs. nested decompositions . . . 39
3.3 The MoJo Distance Metric . . . 41
3.4 The heuristic algorithm . . . 45
3.4.1 Tag assignment . . . 47
3.4.2 Partner assignment . . . 48
3.4.3 Partner refining . . . 49
3.4.4 Partner splitting . . . 51
3.4.5 Profitable joins . . . 51
3.4.6 Move operations . . . 54
3.5 Algorithm analysis . . . 55
3.5.1 Commutativity of *Move* and *Join* operations . . . 55
3.5.2 Order of performing *Join* operations . . . 57
3.5.3 Partner cluster selection . . . 58
3.6 The experiments . . . 59
3.6.1 Performance evaluation . . . 59

3.6.2 Accuracy evaluation 62
3.6.3 Clustering quality metric 62
3.6.4 Parameter tuning 64

4 Stability of clustering algorithms **67**
4.1 Introduction 67
4.2 Definitions 68
4.3 A stability measure 72
4.4 Implementation and Experiments 75
4.5 Conclusions 84

5 Clustering evolution **85**
5.1 The Orphan Adoption problem 86
5.2 The Orphan Adoption Algorithm 89
5.2.1 Overview 89
5.2.2 Structural Criteria 92
5.2.3 Tie-breakers 93
5.2.4 Interface Minimization 94
5.2.5 Observations 95
5.3 The Case Studies 97
5.3.1 Case study 1 : MiniTunis 98
5.3.2 Case study 2 : Neuma 99
5.3.3 Case study 3 : TOBEY 100
5.3.4 Observations 100

6 Pattern-driven clustering **103**

6.1 Introduction . . . 104

6.2 Clustering for comprehension . . . 105

6.3 Subsystem patterns . . . 107

6.4 The ACDC algorithm . . . 117

6.4.1 Stage 1: Skeleton construction . . . 117

6.4.2 Stage 2: Orphan Adoption . . . 121

6.4.3 Algorithm properties . . . 122

6.5 Validation of ACDC . . . 124

6.5.1 Performance . . . 125

6.5.2 Stability . . . 125

6.5.3 Skeleton size . . . 127

6.5.4 Quality . . . 127

6.5.5 Observations . . . 128

7 Conclusions **131**

7.1 Research contributions . . . 132

7.2 Suggestions for extending this research . . . 133

7.3 Opportunities for further research . . . 135

7.4 Closing remarks . . . 137

A Example software systems **138**

A.1 TOBEY . . . 138

A.2 Linux . . . 139

B Software clustering toolkits **143**

B.1 HCAS 144

B.2 Bunch 145

List of Figures

2.1 An example partition sequence 26

2.2 The dendrogram for the example partition sequence 27

3.1 The containment tree of a flat decomposition. The nodes labeled A_i are clusters, while the leaves of the tree are the objects. 40

3.2 Conversion of a nested decomposition to a compact flat decomposition. . 41

3.3 Conversion of a nested decomposition to a detailed flat one. 42

3.4 Two partitions that demonstrate the non-symmetrical behaviour of the *mno* function. 43

3.5 Three partitions that demonstrate the reason for the disallowance of the *Split* operation. 43

3.6 Example partition A comprising 25 objects. 46

3.7 Example partition B comprising the same 25 objects as partition A. . . . 47

3.8 Partition A with the tags. 47

3.9 Partition A containing only tags. 48

3.10 Partition A after partner clusters have been assigned. 49

3.11 Partition A after the partner refining stage. 51

3.12 Partition A containing distinct partner clusters. 52

3.13 Even though cluster A_3 is P_3, the cluster contains more objects tagged with T_4 than ones tagged with T_3 ($w_{34} > w_{33} + 1$). This indicates that it is profitable to join clusters P_3 and P_4. 53

3.14 Partition A after all profitable *Join* operations have been performed. . . . 54

3.15 Partition A at the end of the execution of the HAM algorithm. 54

3.16 An example that demonstrates the efficiency of the order of *Join* operations that HAM selects. 57

3.17 A graphic depiction of the performance experiment results. Field data are shown as green dots, while random data are red. 61

4.1 When a software system is slightly modified, a software clustering algorithm should produce similar results. 69

4.2 Stability values of SL65 and SL75. 79

4.3 Stability values of SL90 and CL65. 80

4.4 Stability values of CL75 and CL90. 81

4.5 Node degree distribution for TOBEY (thick line) and Linux (thin line). A point's x-value is the node degree (total number of incoming and outgoing edges) and the y-value is the number of nodes in the system's graph that have this degree. 83

5.1 The flowchart of the Orphan Adoption algorithm. 91

5.2 The system decomposition of the MiniTunis operating system. 98

6.1 The source file pattern. 109

6.2 The directory structure pattern. 110

6.3 The body-header pattern. 111

6.4 The leaf collection pattern. 112

6.5 The support library pattern. 113

6.6 The Central dispatcher pattern. 114

6.7 The subgraph dominator pattern. Note *z.c* is not included in the pattern instance, since there is a path to it that does not go through *dominator.c* 116

6.8 An example of nested subgraph dominator pattern instances. Node d will be examined before node a, and a cluster called "d.ss" containing nodes d, f, and g will be formed. When node a is examined, a cluster called "a.ss" will be formed containing nodes a, b, c, e, and the previously formed subsystem. 123

6.9 Stability diagrams for ACDC with respect to TOBEY and Linux. 126

A.1 The top level view of the TOBEY Software Bookshelf (dependencies have been omitted to avoid cluttering the picture. 140

A.2 The top level view of the Linux Software Bookshelf. 142

Chapter 1

Introduction

Software developers must be able to understand the structure of the software system they are working with in order to be able to make effective modifications and improvements. However, the sheer size and complexity of most industrial software systems makes this a difficult task. The problem is often exacerbated by the fact that the documentation of the software system is rarely updated, while key developers, knowledgeable of the system's structure, commonly move to new projects or companies.

The focus of this dissertation is on developing techniques that can facilitate the task of recovering the structure of software systems in an automatic way. Before describing our work in detail, we first present an introduction to the software clustering problem.

1.1 The software clustering problem

Software Reengineering

Industrial software systems commonly have two properties that make the understanding of their structure a difficult task: they are large and complex. A typical system consists of thousands of entities (procedures, classes, modules) that are interconnected in various ways (procedure calls, inheritance relations). It is common that the source code documentation is either obsolete or non-existent. This means that only the system architects and/or seasoned developers might know the software system's structure. This knowledge is often lost when these knowledgeable people are transferred to different projects or find employment in a different company. For many legacy software systems that are currently in use around the world, this knowledge was lost years ago.

The abundance of software reengineering projects indicates that the need to modify existing industrial software systems arises rather frequently. Even now that the Y2K problem is behind us, there are many reasons why modifications might be necessary. These include migration to new hardware platforms or operating systems, compliance with new industry standards, or a change in the requirements.

The research field of software reengineering attempts to deal with the difficult problems of understanding, redocumenting, and modifying software systems. One of the main factors that makes these problems tough to address is the sheer size of these software systems.

Software Clustering

Researchers in other disciplines, such as biology, social sciences, and engineering, have also been faced with large amounts of data. An approach that is commonly employed to address this complexity is to develop a taxonomy, i.e. create categories of objects that exhibit similar features or properties. Such categories (commonly referred to as *clusters*) can be discovered through a variety of techniques that have been proposed in the literature. Research on the effectiveness and behaviour of these techniques has given rise to the field of *cluster analysis* [And73, Har75, Eve93, JD88, Zup82].

Several software researchers have developed similar techniques in a software context, either by adapting existing cluster analysis techniques, or by introducing new ones. The common goal of all the approaches presented in the literature is this: to decompose large software systems into smaller, more manageable subsystems that are easier to understand. Such techniques are collectively referred to as *software clustering* techniques.

A variety of software clustering approaches has been presented in the literature. Each of these approaches looks at the software clustering problem from a different angle, by either trying to compute a measure of similarity between software objects[SP89]; deducing clusters from file and procedure names[AL97]; utilizing the connectivity between software objects[HB85, Nei96, CS90]; or looking at the problem at hand as an optimization problem[MMR+98].

Our Emphasis on Comprehension

The software clustering approaches that have been presented in the literature can be categorized in three different classes depending on the goal that they strive to attain:

1. *Remodularization approaches.* Their goal is to remodularize (group the system's resources in a different way) in order to improve certain properties of the software system such as its maintainability or evolvability.

2. *Objectification approaches.* Several software reengineering projects involve the migration of the source code from a procedural language to an object-oriented one. These approaches attempt to identify collections of entities that would be candidates for classes.

3. *Comprehension approaches.* Their goal is to decompose the software system into smaller, more manageable subsystems in a way that will aid the task of understanding the structure of the software system.

The emphasis of this thesis is on software clustering approaches that have comprehension as their main goal. For convenience, the term software clustering approach will be used to refer to an approach geared towards comprehension.

Our Emphasis on Procedural Software Systems

The majority of the software clustering approaches found in the literature are targeting software systems written in procedural programming languages. The fact that the bulk of the legacy software is written in such a language is only one of the reasons for this phenomenon.

This thesis will assume that the software systems in question are written in procedural languages for the following reasons:

- It is easier to find example software systems written in a procedural language.

- Comparison to other software clustering approaches is possible, since they also target the same type of systems.

- Parsers for common procedural languages are readily available, while this is not necessarily true for languages such as C++ or Java.

- The procedural domain model used by most researchers is standard (procedure calls, data and type references). However, there is no consensus as to which relationships should be used when dealing with an object-oriented system.

Software Clustering as an Emerging Research Area

The proliferation of approaches presented in the software clustering literature in the recent years indicates that the software clustering problem has been receiving much attention lately. The importance that the software engineering community assigns to the software clustering problem is indicated by the large number of publications on the subject that appear in the proceedings of many conferences, such as the Working Conference on Reverse Engineering, the International Conference on Software Maintenance, and the International Workshop on Program Comprehension. The International Conference on Software Engineering has also devoted several sessions to this research issue. The techniques published in these forums view the software clustering problem from a variety of angles. The following presents major families of software clustering approaches:

- *Name-based.* These techniques determine which entities should belong in the same subsystem by the similarity of their names [AL97]. Several name-based techniques take into account also any comments that may reside in the source code [MMM93].

- *Feature-based.* A number of features is associated with each entity. These features range from the entity's size to a list of dependent entities. Various techniques such as agglomerative clustering algorithms [AL99] or concept analysis [LS97] are then employed to categorize the entities based on the correlation of these features.

- *Structure-based.* These techniques use dependencies between entities to "discover" subsystems. Various kinds of dependencies, such as data bindings [HB85], procedure calls, and data references, have been utilized, and a variety of criteria, such as high cohesion and low coupling [CS90], shared neighbours [SP89], and few/small interfaces [MOTU93], have been employed for this purpose.

- *Miscellaneous.* Finally, there exist several techniques that adapt methods from other disciplines, such as latent semantic analysis [MV99] and genetic algorithms [MMR+98].

Software Clustering is not a "Silver Bullet"

Decomposing a software system is an inherently difficult task. There is a consensus between software clustering researchers that a software clustering approach can never hope to cluster a software system as well as an expert who is knowledgeable about the system. An automatic approach can only hope to come close to the desired solution. In this way, the task of the expert will be greatly simplified, since the software clustering technique will create a decomposition that only needs to be refined rather than constructed from scratch.

Therefore, it is fair to say that software clustering techniques are not the "silver bullets" that will save software reengineering projects from the inherent complexity of

their task. However, we believe that they have the potential of improving our ability to understand complex software systems faster and more accurately.

There remain many tasks ahead of us, and many open questions to be answered. In the next section, we discuss several of them.

1.2 Open issues on software clustering

Research on software clustering has been active for almost twenty years. Despite the progress that has been made, the issue is far from resolved. Several of the open questions that researchers on software clustering are facing are discussed in this section.

- *Evaluation of approaches.* An important issue is the evaluation of an approach's usefulness. Most researchers attempt to validate their work by applying it to a number of software systems. However, these systems are almost invariably either of a rather small size or proprietary, making it hard to draw any significant conclusions.

- *Comparison of results.* Another important problem is comparing the results of various approaches. This issue has two distinct facets. Firstly, there exists a need for a collection of software systems that can be used as benchmarks. These systems would have to be of considerable size, well-known structure and ready availability. Secondly, one needs a way to compare the output of various software clustering algorithms on these benchmark systems. When dealing with systems comprised of thousands of source files this task is far from trivial.

- *Stability.* A property of software clustering algorithms that has not attracted much attention is their stability. More effort needs to be put into determining how is

the output of a software clustering algorithm affected when its input is slightly modified. High stability can be a crucial feature for the viability of an algorithm associated with a software system that is modified on a daily basis.

- *Incremental clustering.* A related problem is that of the development of incremental clustering algorithms. Software clustering is commonly applied to software systems that are currently under development. This means that, even if a satisfactory clustering is found, this decomposition will soon be out-of-date. Since it is important to maintain the system's basic structure, there has to be a way to adjust the existing breakdown in order to accommodate the latest changes.

- *Approach focus.* Finally, there is a trend in the literature towards improving the precision (e.g. finding a partition that exhibits a larger value of some cohesion/coupling metric) or the performance (e.g. finding the clustering faster) of existing techniques. While these aspects are necessary, we believe that it is more important to ensure that the obtained decomposition will be helpful to humans attempting to understand the given software system.

In the next section, we present our approach to the issues described above.

1.3 Research contributions

The main research contributions of this dissertation are:

1. The introduction of a metric that measures distance between two partitions of the same set of data.

2. The study of the notion of stability of software clustering algorithms and presentation of a measure that gauges it.

3. The development of an incremental clustering algorithm.

4. The notion that software clustering efforts should be focused on understanding and the development of an automatic clustering algorithm that subscribes to this philosophy.

The remainder of this section elaborates on these four contributions, while in the next section we present an outline of the dissertation.

1.3.1 A distance metric for software clusterings

The main reason why it is hard to validate a software clustering approach is the fact that there exists no easy way to assess the value of a given decomposition. The software system's design architects can do it in a painstaking way by examining all clusters carefully. However, this would be a time-consuming process that is unlikely to fit their schedule.

Some software systems have been clustered manually by their designers. Even when such an authoritative clustering exists, it is usually hard to determine which algorithm's output comes closer to it due to the large number of discrepancies. For this reason, we have developed MoJo, a metric that measures distance between two partitions of the same data set.

MoJo defines this distance as the minimum number of **Mo**ve and **Jo**in operations one needs to perform in order to transform one of the partitions into the other. We have developed a heuristic algorithm that computes this number. This algorithm has

been implemented; we present several experiments that demonstrate the efficiency of our heuristic and the usefulness of the metric.

1.3.2 Stability of software clustering algorithms

One of the biggest challenges a software clustering algorithm faces is that of fitting into the daily implementation cycle of a real-life software development project. In other words, a good way to measure the effectiveness of a software clustering algorithm would be to include it in the daily routine of a developing team. Assuming that the system is built nightly, the software clustering algorithm would produce a new decomposition of the software system that would be shown to the developers the next morning, probably in the form of diagrams produced by a visualization tool. For a software clustering algorithm to be accepted as part of the daily routine, it would have to produce "meaningful" decompositions, something all software clustering algorithms strive to do. However, another factor might also play an important role. The software clustering algorithm will have to adhere to the "rule of minimal change", i.e. the diagrams presented each morning should not be radically different from the ones shown the previous day, since that could lead to frustration, and eventually the rejection of the software clustering algorithm.

This raises an issue that has not attracted much attention from the software clustering community so far: a software clustering algorithm's *stability*. It would be interesting to know how much is the output of a software clustering algorithm affected, when its input is slightly modified. For this reason, we have developed a measure that ascertains whether a software clustering algorithm is stable.

Our stability measure defines in detail what "small changes" to the input mean. We

have performed them in a random way that attempts to simulate the way modification happens in reality. The MoJo metric is utilized in order to measure the effect on the algorithm's output. We have experimented with a variety of software clustering algorithms, and present results that demonstrate the validity of our measure.

1.3.3 Incremental software clustering

An alternative approach to the problem outlined above, i.e. how to handle the effect of software evolution on the structure of a software system, is to develop an algorithm that takes into account the existing decomposition and modifies it in order to accommodate the latest changes.

We identify two interesting subproblems, namely that of assigning a newly introduced resource to a subsystem (incremental clustering), and that of accommodating structural changes (corrective clustering). We propose an algorithm to handle these problems and validate it by presenting a number of case studies. Our algorithm (known as the Orphan Adoption algorithm) has been part of the PBS toolkit [HF98] for a number of years with successful results.

1.3.4 Clustering for comprehension

Software clustering techniques presented in the literature generally employ certain criteria in order to decompose a software system. Such criteria include low coupling and high cohesion, interface minimization, shared neighbours etc. However, it appears that in an effort to maximize the performance or accuracy of their algorithms, many researchers have lost sight of the fact that their primary concern is comprehension. Our first priority should

not be to ensure that the clusters produced by our approach satisfy certain abstract criteria, but that they can be effective in helping someone understand the software system.

For this reason, we believe that an effective software clustering algorithm should produce output that can be easily understood, by proposing clusters that follow familiar patterns, and by naming these clusters appropriately. Also, the size of the obtained clusters can be an important factor. A cluster containing more than about a hundred objects is not very useful, even though it might exhibit a high degree of cohesion. We should strive to keep the size of the clusters at a more manageable level (up to 20 objects or so), by creating hierarchies of nested clusters. Coupled with effective visualization techniques, such an approach can produce clusterings that are actually helpful for the comprehension of a software system.

We present a software clustering algorithm that subscribes to this philosophy. It discovers clusters that follow patterns that are commonly observed in decompositions of large software systems that were prepared manually by their system architects. It also automatically assigns meaningful names to the clusters and tends to avoid creating clusters of very large size. The name of our algorithm is ACDC (for an **A**lgorithm for **C**omprehension-**D**riven **C**lustering). Several experiments have been conducted with large industrial systems, and their results are presented to demonstrate the usefulness of our algorithm.

1.4 Dissertation outline

- **Chapter 1 - Introduction**

  This chapter introduces the software clustering problem and identifies some of the

open questions in the area. It proceeds with an introduction to our approach to addressing these issues. The chapter concludes with an outline of the dissertation.

- **Chapter 2 - Background**

  This chapter begins with a historical overview of various approaches to the software clustering problem. It then presents some of the latest ideas on this subject. This is followed by a survey of the more important cluster analysis techniques. Similarities and differences between cluster analysis and software clustering techniques are then outlined. The chapter concludes with a presentation of research challenges in the area.

- **Chapter 3 - Comparison of clustering techniques**

  This chapter outlines the rationale behind our MoJo metric, describes the details of our heuristic algorithm, and presents experiments we have conducted.

- **Chapter 4 - Stability of clustering algorithms**

  In this chapter we present our approach to studying the stability of software clustering algorithms. We define a measure that assesses if a given algorithm is stable or not. Experiments with a variety of clustering algorithms demonstrate the validity of our measure.

- **Chapter 5 - Clustering evolution**

  Our algorithm that performs incremental and corrective clustering is presented in this chapter. It is known as the Orphan Adoption algorithm (an *orphan* is a newly introduced resource to an existing software system). Experimental results are used to demonstrate the algorithm's usefulness.

- **Chapter 6 - Pattern-driven clustering**

  This chapter begins by presenting the features a software clustering algorithm should have in order to be helpful to a program comprehension project. It continues by presenting ACDC, an algorithm that encompasses these features. ACDC is pattern-driven, i.e. it identifies subsystems that conform to patterns commonly observed in large software systems. The chapter concludes by applying ACDC to a number of such systems in order to evaluate its usefulness.

- **Chapter 7 - Conclusions**

  This final chapter summarizes the contributions of this work and indicates directions for further research.

Chapter 2

Background

2.1 Introduction

The definition of the term "large software" is constantly changing, as the size of software systems continues to increase rapidly. What DeRemer and Kron called a "large system" in their classic paper on Programming-in-the-Large [DK76], would probably be classified as a medium-sized, if not small-sized, system today. Advances in hardware technology concerning the size, speed, and cost of primary and secondary storage, as well as the advent of modern programming languages and object-oriented programming, have allowed the size of software systems to increase significantly.

When a system becomes large, it is very hard to ensure that its structure is the intended one. Moreover, the original documentation, if it exists at all, becomes outdated as the system evolves, since the developers are usually busy trying to meet the next deadline. The fact that developers often discontinue their association with such large projects intensifies the problem, since they take a lot of the knowledge about the system

with them.

These factors contribute to the transformation of a piece of software into what is known as "legacy code", namely a piece of code that one uses but does not necessarily understand. The drawbacks of having legacy code in a software system become obvious when the time comes to alter its functionality, to adapt it to a new hardware platform or operating system, or to improve its performance. One needs to understand the code once again.

Even systems that are still under development are impacted by these problems. Parts of the system might become legacy code, if only because they have not been maintained for some time. Also, large projects often hire new people who must be brought up to speed, but the seasoned developers are often too busy to help with this. If the documentation is obsolete, then a newcomer is at a loss as to where to start, and cannot know the full impact of a potential modification on the rest of the system.

Clearly, a solution to all these problems is required. If one could derive a decomposition of a large software system into meaningful subsystems in an automatic (or semi-automatic) way, then much of the effort required to understand and to improve a software system would be alleviated. At the same time, this capability would help one to remodularize legacy code, as well as to identify candidate subsystems for extraction of reusable components.

Automatic software clustering techniques described in the literature claim that they can do exactly this — detect the "natural" groups (or *clusters*) in a collection of entities, such as procedures or source files. In this chapter, we will discuss the most important software clustering approaches that have been presented in the literature.

Furthermore, since research on clustering began long before the term "software" was

coined, many techniques have been developed by researchers in other disciplines. This chapter will also present a survey of the more common *cluster analysis* techniques, as well as discuss the possibility of reusing these techniques in a software context by adapting them to fit the peculiarities of this specific problem domain. A presentation of several crucial issues related to software clustering will conclude this chapter.

Clustering Terminology

A considerable confusion of terminology has resulted from the fact that clustering techniques have been used in a variety of disciplines. The raw material to be clustered has been called "point", "item", "data unit", "subject", "object", "element", "entity" and many other terms. This dissertation will use the terms *resource*, *node*, and *object* interchangeably depending on whether we're looking at it from a software engineering, graph theory, or cluster analysis point of view respectively. Also, the aspects of the objects that we look at in order to decide on the appropriate clustering have been called "variables", "attributes", "characters", or "features". We will use the term "feature".

Moreover, the terms *clustering*, *partition*, and *decomposition* will all be used to describe the output of a software clustering algorithm. The term "clustering" usually implies that it is possible for an object to belong to more than one cluster. However, software clustering techniques rarely operate in such a fashion, which allows us to use the terms "clustering" and "partition" interchangeably.

2.2 Previous Work on Software Clustering

2.2.1 Knowledge-based vs. Structure-based approaches

A common approach to the problem of understanding a software system and recovering its design is the *knowledge-based* approach. In the *bottom-up* version of this approach, one attempts to reverse-engineer and understand small fragments of the source code, using pre-existing domain knowledge. One then combines these fragments together in an effort to understand the system as a whole, thus determining its design. This approach has been shown to work well with small systems.

When dealing with large systems, however, this approach does not perform as effectively. The size of the knowledge base becoming prohibitively large is one of the reasons. Other reasons are outlined by Neighbors [Nei96]: "Knowledge-based understanding of large system semantics [is] currently too difficult for three reasons: absence of a robust semantic theory, lack of problem domain specific semantics, and knowledge spreading in the source code".

For these reasons, the software clustering community mainly adopts *structure-based* approaches. In these approaches, the decomposition of a software system is determined by looking at syntactic interactions (such as "call" or "fetch") between entities (such as procedures or variables). In this case, the problem of clustering a software system can be thought of as the partitioning of the vertex set of a graph, where the nodes are defined as program entities (usually procedures or source files), and the edges as dependencies between these entities.

The rest of this section presents a survey of important publications in the field of software clustering, as well as some of the recent approaches to the problem.

2.2.2 Key publications

In one of the early works in software clustering, L. A. Belady and C. J. Evangelisti [BE81] recognized the need to automatically cluster a software system in order to reduce its complexity. They also presented a first approach to doing this for a specific system. In addition, they provided a measure for the complexity[1] of a system after it has been clustered. Their approach, however, only works with a specific kind of system, and they did not validate their complexity measure. A point of interest is that they didn't extract information from the source code, but rather from the system's documentation.

Subsequent to this work, Hutchens and Basili [HB85] performed clustering based on *data bindings*. A data binding was defined as an interaction between two procedures based on the location of variables that are within the static scope of both procedures. They defined different kinds of data bindings; from simplistic and easy to compute (e.g. a data binding between two procedures p and q exists, if there exists a variable that belongs in the static scope of both procedures) to sophisticated and hard to compute (e.g. the data binding only exists if control flow might be given to procedure q after the value of the common variable has been set by p). On the basis of the data bindings, a hierarchy is constructed from which a partition could be derived.

An interesting feature of their paper is that they compared their structures with the developer's mental model with satisfactory results. They also raised the important issue of *stability*; when the system changes slightly, how is the clustering affected? Finally, they recognized that it might be necessary to disregard certain information, such as omnipresent nodes, in order to get a clearer view of the structure of a software system.

[1] Complexity here refers to how difficult it is to understand a system after it has been clustered in a specific way.

One of the most active researchers in the area of software clustering in the early 1990s was Robert W. Schwanke. His papers [SAP89, SP89, Sch91] and his tool (called ARCH) addressed the problem of automatic clustering in an innovative way. Although his approaches were not tested against a large software system, they showed considerable promise.

One of his main contributions was that he added to the "classic" low-coupling and high-cohesion[2] heuristics by introducing the "shared neighbors" technique [SP89] in order to capture patterns that appear commonly in software systems. This refers to identifying subsystems that are not comprised of resources cooperating to implement a specific functionality, but rather of resources providing similar functionality, such as the routines of a math library.

Furthermore, his "maverick analysis" [Sch91] enabled him to refine a partition by identifying components that happened to belong to the wrong subsystem, and placing them in the correct one. He also attempted to provide names and descriptions for automatically generated clusters, but not very convincingly.

Choi and Scacchi [CS90] presented an approach to finding subsystem hierarchies based on resource exchanges between modules. The complexity of their algorithm is $O(n^2)$, which is better than Schwanke's $O(n^3)$, but still probably too high for large systems. It appears to perform well on small examples, but its ability to scale up is questionable.

Hausi Müller has also been involved in the automatic clustering problem [MU90, MOTU93]. His approaches tend to be semi-automatic, meaning that they are meant to

[2] *Low coupling* is a software engineering principle that requires that interactions between subsystems should be as few as possible. *High cohesion* is a related principle that requires that interactions should be kept as much as possible within a subsystem.

help a designer perform clustering on a software system. He introduces the important principles of *small interfaces* (the number of elements of a subsystem that interface with other subsystems should be small compared to the total number of elements in the subsystem) and of *few interfaces* (a given subsystem should interface only with a small number of the other subsystems).

2.2.3 Recent work

The last couple of years have seen a renewed interest in the problem of clustering a software system automatically. The main reason for this is the rapid growth of reverse engineering as a research field, partly due to the Year 2000 and Euro conversion[3] problems. Understanding large software has become a very important issue and clustering can help deal with it.

Arun Lakhotia introduced a unified framework for expressing software clustering techniques [Lak97]. Realizing that the techniques in the software clustering literature have been presented using different terminology and symbols, he proposed a framework consisting of a consistent set of terminology, notation, and symbols, that can be used to describe the input, output, and processing performed by these techniques. Several existing techniques were reformulated to conform to this framework.

James M. Neighbors attempted to identify subsystems with the ultimate goal of hand extraction of reusable components [Nei96]. He looked at compile-time and link-time interconnections between components and tried different approaches. The approaches that were successful were based on naming and on reference context. His results were

[3]Financial software in Europe had to be modified in order to accommodate the common currency.

validated by the developers of the system on which he experimented.

An interesting alternative approach was presented by Nicolas Anquetil and Timothy Lethbridge [AL97]. Instead of looking at structural information, such as procedure calls or data references, they only looked at the names of the resources of the system. Their experiments produced promising results, but their approach has the obvious drawback that it relies on the developers' consistency with the naming of their resources.

Ivan Bowman and R.C. Holt introduced the term *ownership architecture* [BH99]. They argued that the organization of system developers into teams can help one understand a software system, and that such a structure is often congruent to the system's concrete architecture. An extensive case study involving the Linux operating system was presented to corroborate their conjecture.

Finally, various researchers have recently started looking at techniques used in other disciplines in order to come up with a better solution to the automatic clustering problem.

Theo Wiggerts [Wig97] presented a survey of techniques used by the cluster analysis community and attempted to reuse them for system remodularization. His future plans included clustering a software system in a "more or less object-oriented" way.

Several researchers attempted to use concept analysis in order to identify subsystems [LS97, vDK99]. Their experiments demonstrate that concept analysis could be helpful in certain reverse engineering scenaria, such as object identification.

Spiros Mancoridis [MMR+98] treated clustering as an optimization problem and employed genetic algorithms in order to overcome the local optima problem of "hill-climbing" algorithms, which are commonly used in clustering problems. His experiments demonstrate encouraging results and fast performance.

2.2.4 Observations on the software clustering literature

By examining the literature on software clustering one can draw interesting observations. First, most researchers seem to agree on structural-based criteria and naming conventions as being the most promising approaches. However, there exists a variety of different interactions between modules that are used as the basis to decide which resources depend on which. Isolating the interactions that are appropriate for the software clustering problem, and determining the properties that make them so, is a problem that needs more study.

Another observation is that none of the approaches has been tested extensively against large software systems[4]. This omission becomes more interesting when one considers that these approaches were developed with such systems specifically in mind. It is not clear whether these approaches scale up to large systems.

Also, validation of an approach against more than one system is required. Many researchers present results that demonstrate that their algorithm performs very well for a given software system. It would be interesting to see how the algorithm performs on a number of systems, since an algorithm can be specifically tuned to perform well on a particular system.

Finally, the issue of performance is important. Most graph partitioning problems are shown to be NP-complete [GJ79] or NP-hard. The approaches presented above, however, are heuristic approaches that attempt to reduce this complexity to polynomial upper bounds. The kind of complexity that is acceptable for large systems remains to be seen.

[4] A large system refers to an industrial system with a size of order of magnitude close to a million lines of code.

A more detailed description of open problems and research challenges can be found in section 2.4. In the next section, we present popular cluster analysis approaches that have been used to solve clustering problems found in other disciplines.

2.3 Cluster Analysis Techniques

Cluster analysis has been used in a number of different disciplines such as psychology, biology, statistics, social sciences, and various fields of engineering, in order to solve a wide spectrum of problems. Its objective is to find algorithms and methods for grouping or classifying objects. Many diverse techniques have been developed in order to discover structure within complex bodies of data.

In this section, we will present the most important cluster analysis techniques found in the literature.

2.3.1 Similarity Measures

One of the first things that a clustering approach usually does is to decide on what grounds two objects will be judged to be similar. Moreover, one needs a measure that will decide which pair of objects are "more similar" than any other pair. The answer to this problem is a *similarity measure*.

Similarity measures can be divided in two groups, depending on the kind of information that serves as their input. We distinguish the following kinds of information:

1. *Relations between the objects.* In this case, the problem can be represented as a graph, where the nodes are the objects and the edges are the relations. If we have more than one relation, then the graph will have multiple kinds of edges.

Common similarity measures that deal with cases like this are based on the number of edges connecting two objects, the length of the shortest path between two objects, or the weight that different kinds of edges might have. Whether the graph is directed or undirected is also a factor.

2. *The score of the objects on different features.* In this case, similarity is usually measured by *association coefficients*. These are expressed in terms of the number of features which are present for each object. For this reason, association coefficients assume binary features (i.e. reflecting whether a feature is either present or not). The following table is used in order to calculate various coefficients between object i and object j:

	Object j **1**	Object j **0**
Object i **1**	a	b
Object i **0**	c	d

In the above table, a is the number of features that are present for both objects, b the number of features present only for object i, and so on. Different coefficients treat 0-0 matches (their number is given by d) differently and also put different weightings on any of the four entries of the table. The most common coefficients are:

- the *simple matching coefficient*, defined as: $\frac{a+d}{a+b+c+d}$
- the *Jaccard coefficient*, defined as: $\frac{a}{a+b+c}$

An extensive study of coefficients can be found elsewhere [And73].

Other similarity measures that are found in the literature include distance measures (usually Euclidean or Manhattan), correlation coefficients, and probabilistic measures (based on the assumption that agreement on rare features is more important than agreement on frequent ones).

2.3.2 Algorithms

Once the similarity measure has been decided upon, an appropriate algorithm has to be chosen as well. The majority of the algorithms found in the literature can be categorized into one of the following three categories : *hierarchical* algorithms, *partitional* algorithms, and *graph-based* algorithms. We will present each category in detail.

Hierarchical algorithms

These algorithms produce a nested sequence of partitions. At one end of this sequence is the partition where each object is in a different cluster (we will call this partition ALL) and at the other end the partition where all the objects are in the same cluster (we will call this partition ONE). At each step through this sequence two of the clusters are joined together. Figure 2.1 shows an example partition sequence for four objects A,B,C, and D.

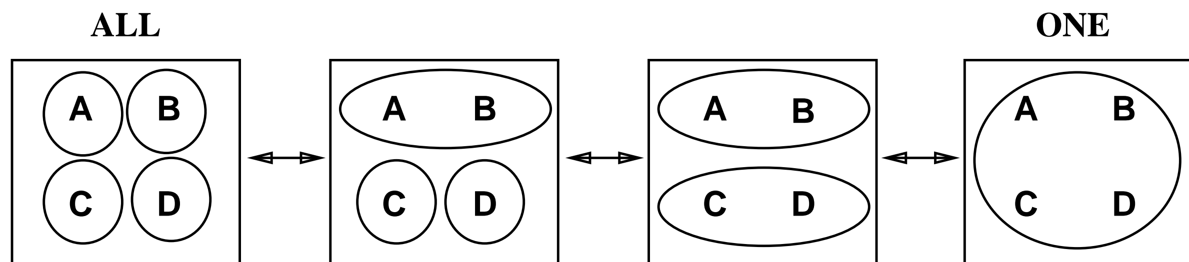


Figure 2.1: An example partition sequence

A common representation for a hierarchical structure is that of a “dendrogram”. Figure 2.2 presents the dendrogram for the example partition sequence.

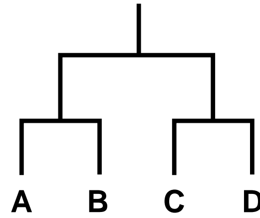


Figure 2.2: The dendrogram for the example partition sequence

However, different partitions in a sequence like the one shown in figure 2.1 are not of equal importance. Actually, only a small number of them (maybe only one) is usually interesting. This is usually referred to as finding “cut points” for the dendrogram. Factors that influence the selection of a “cut point” are usually a priori knowledge on the expected structure, or prechosen parameters such as the maximum number of clusters allowed, or the maximum number of objects in a cluster.

Hierarchical algorithms are divided into two categories, agglomerative and divisive:

1. *Agglomerative (or bottom-up).* These start with partition ALL and iteratively join the most similar clusters based on the similarity measure. An interesting point of debate between researchers is how to compute similarity between a newly-formed cluster and the rest of the already formed clusters. This is called the *update rule* problem.

   Many different solutions exist for it. The most common include the single-link update rule (the similarity of the newly formed cluster to an existing cluster C is the *maximum* of the similarities of its constituents to C), and the complete-link

update rule (the similarity of the newly formed cluster to an existing cluster C is the *minimum* of the similarities of its constituents to C).

2. *Divisive (or top-down).* These start with partition ONE and try to iteratively split it until we reach partition ALL. Such algorithms suffer from excessive computational complexity, as there is an exponential number of possible partitions at every step. This is the main reason why these algorithms are not very popular.

Partitional algorithms

Partitional algorithms usually work by starting with an initial partition and trying to modify it in an attempt to optimize a criterion that represents the quality of a given partition. What constitutes a good criterion is an interesting problem. It usually has to do with the domain on which one tries to cluster. For example, a criterion could be a mathematical expression that is maximized when the cohesion of the clusters is maximized.

The challenge that partitional algorithms face is the combinatorial explosion of the number of possible partitions. Even for a small number of objects the number of possible partitions is astronomical. For example, there are 34,105 partitions of ten objects into four clusters, but this number explodes to approximately 11,259,666,000 if the number of objects is increased to 19.

The usual workaround to this problem is to start with an initial partition (chosen randomly or based on some heuristics) and attempt to optimize the chosen criterion by modifying that partition in an appropriate way. These algorithms (called hill-climbing algorithms) do converge [And73], but usually to local optima. Therefore, the choice of

the initial partition is crucial for the success of the algorithm.

Perhaps the best-known partitional algorithm is ISODATA [And73]. Its effectiveness is based on the successful initial choice of value for seven parameters that control factors such as the number of expected clusters, the number of objects in a cluster, etc. The algorithm then proceeds to iteratively improve on an initial partition by joining and splitting clusters, depending on how close to the chosen parameters the actual values for the current partition are. Several variations of this method exist in the literature.

Graph-based algorithms

A particular class of algorithms that are of interest are the ones that are based on graph properties. Different categorizations of these techniques exist, depending on the perspective one chooses [CW78, Wig97]. We distinguish the following categories:

- *Minimum Spanning Tree (MST) algorithms.* These algorithms begin by finding an MST of the given graph. Next, they either iteratively join the two closest nodes into a cluster (hierarchical agglomerative version) or split the graph into clusters by removing "inconsistent" edges (partitional version). The definition of an inconsistent edge varies, but they are usually of considerably larger weight than the rest of the edges of the MST.

- *Clique algorithms.* These algorithms start with the maximal complete subgraphs (cliques) of the given graph, and either define them as clusters, or use them as the basis for other algorithms.

- *Local connectivity algorithms.* The number of edge or vertex disjoint paths of a specified length between two points is the criterion used in these algorithms in

order to decide which objects (or nodes in this case) are similar enough to be in the same cluster. For example, instead of only using edges (paths of length 1) as an indication of "closeness", one could also use paths of length 2 [Vas68].

- *Aggregation algorithms.* These algorithms select sets of nodes and collapse them into aggregate nodes, which can be used as clusters or can be input for a new iteration to find higher level aggregates. Graph reduction is an aggregation technique that is based on the notion of the neighbourhood of a node [vL93]. Bi-components and strongly connected components have also been used for this purpose [BS91].

- *Heuristic approaches.* As mentioned before, the large number of possible partitions makes the problem of graph partitioning almost impossible to solve optimally. Heuristic approaches attempt to search the space of possible solutions in a clever way in order to come as close to the optimal solution as possible in a reasonable amount of time. For example, the Kernighan-Lin method [KL70] attempts to overcome the local optima problem of hill-climbing algorithms by choosing to go "downhill" for a while in the hope of finding a taller "hill" in the next few steps.

2.3.3 Observations

By examining the literature on cluster analysis, one can draw some interesting observations. First, researchers agree that "a classification is neither true or false" [Eve93]. This means that no particular partition can be the ideal answer to the problem of classifying a large number of objects. Based on different points of view, one can come up with two different, but equally valid, decompositions of the same set of objects. Some classifications, however, are more useful than others. It is the job of the cluster analysis researcher

to find what factors determine the usefulness of a particular clustering.

Another observation is that "the multitude of alternatives makes it difficult to say that a particular measure and a specific method are clearly superior selections for treating the problem at hand" [And73]. On any given problem, a large group of different methods will give practically the same results, while perhaps a few other methods will give distinctively different results. A theoretic explanation of the behaviour of different methods does not exist, however.

Finally, looking back at the literature on software clustering, we see that some cluster analysis techniques have indeed been used by software researchers. Hutchens and Basili [HB85] used a hierarchical agglomerative method in order to perform their clustering. Schwanke [SP89] defined binary features in the same way as defined earlier in this section. Mancoridis [MMR+98] used a hill-climbing optimization approach similar to the one presented in section 2.3.2.

It remains to be seen whether the software community will adopt more cluster analysis techniques. There are many reasons that indicate that this might be a good idea. One of them is that the peculiarities of software as a clustering domain could be used to alleviate a lot of the problems cluster analysis techniques have to face. For example, with software we already have a rather good idea of what a cluster should look like. Software Engineering principles such as *information hiding* [Par72], or *few interfaces* could guide the clustering process towards a desirable solution. Also, since our goal is usually to understand a software system, we can cluster to different levels of proximity[5] depending on our perspective. Therefore, specifying the number of clusters is not a big

[5]Proximity, in the cluster analysis literature, refers to how close to the data we look, i.e. are we trying to find a few or a lot of clusters.

problem for software clustering. Furthermore, a software system can have more than one valid view, which is what different clustering algorithms can give us.

Another interesting issue in the cluster analysis literature is that of "clustering tendency". Most clustering algorithms can be "accused" of imposing a structure on a set of data, even if no structure exists. In this case, it is possible that the structure presented as the final solution is an artifact of the algorithm used, rather than a "natural" grouping of the objects in question. With legacy software however, this need not be a problem. As it has been noted [Wig97], in many cases any structure is better than no structure, since one needs to start somewhere. Besides, one would hope that even the most badly written piece of code would have some structure.

In their classic text on "Algorithms for Clustering Data" [JD88], Jain and Dubes present a framework for a cluster analysis project, which is divided into 7 steps. This framework seems to fit the software clustering problem as well, as demonstrated by the following adaptation of the original framework, where the original title of each step is accompanied by its elaboration in a software context:

1. *Data collection.* This refers to extracting the relevant information from the source code. A critical issue here is what kind of information one needs to extract.

2. *Initial screening.* The data extracted from the source code usually requires some massaging before it can be used. As noted in [HB85], certain information may have to be deleted as it might interfere with the clustering process, e.g. omnipresent nodes[6] [MU90].

[6]This refers to nodes (typically procedures in the software case) with a large in- or out-degree. In the software case, this might correspond to library routines, the interactions with which are not necessary in order to decide on the structure of the rest of the system.

3. *Representation.* This refers to choosing the appropriate similarity measure. It is usually based on the type of information available, the experience of the investigator, and the insight of system experts. In the case of software clustering, there exists a wealth of different software metrics [AB87, Kon97] that could be used for this purpose.

4. *Clustering tendency.* This step checks if the available data have a natural tendency to cluster or not. As explained before, this is not usually a problem in a software context.

5. *Clustering strategy.* The algorithm to be used and the value of any parameters in it are chosen during this step. It is up to the investigator to decide on the most appropriate algorithm. Comparative studies between different existing algorithms would facilitate the process of choosing or developing effective algorithms.

6. *Validation.* Formal techniques for the validation of a partition exist, but in a software context there are usually alternative methods available. Developers associated with the examined software project can compare the partition obtained from the automatic clustering approach with their own mental model of the structure of a system. Also, in the case of legacy software, empirical studies could evaluate whether the clustering actually helped in the understanding of the system.

7. *Interpretation.* This refers to comparing results with other studies, drawing conclusions, and getting ideas for improvements on any of the previous steps.

In the next section, we will present open problems and research challenges a researcher in the field of software clustering might have to face.

2.4 Research Challenges

Several open problems facing researchers in software clustering have already been presented in section 1.2. In this section, we present a list of further open problems that pose interesting challenges to the researchers in the area:

- It is not clear which kinds of relations between "software objects" are appropriate from a clustering point of view. Procedure calls and data references are commonly used, but what about relations such as source inclusion [CTH95] or type references? Should they be used, and if so, with equal weight to other relations or not? The field is in need of a "theory of dependencies" that characterizes such relations.

- Another interesting research issue is the selection of appropriate algorithms. A comparative study tested on a number of systems is long overdue. It is possible that certain algorithms are best suited for a particular type of software system. A categorization of algorithms and the types of software for which they work best would be beneficial to the software clustering field.

- There exists a gap between the structures obtained by the software clustering researchers and the ones presented by the software architecture community [GS93, SG96]. Closing this gap is not easy, as a compromise has to be found between the automatic approaches of the clustering community, and the "supervised" ones of the software architecture community [HRY95].

- Clustering approaches need to be tested on large systems, as success on small systems does not guarantee effective scaling up to large systems. Obtaining access to large systems is not easy, but it can be done, and it is crucial for the validation of

candidate approaches. Many open source systems appear to be promising candidates for the creation of a benchmark for different algorithms. An updated version of the proposed subject programs is needed [LG95].

- Most software approaches currently present static views of the structure of a software system. However, most large systems are complex enough to require more elaborate views, such as dynamic views. This is certainly an important challenge for the software clustering community.

- Software is a peculiar clustering domain since the developers that are associated with the system being examined can provide a lot of help. Integrating information obtained from the developers with the automatic approaches described in this paper is an important challenge.

The aforementioned problems pose interesting challenges to researchers, and suggest that the software clustering field is a fertile one for research.

2.5 Conclusion

The goal of this chapter was threefold:

- To present the state of the art in the research of software clustering.

- To survey cluster analysis techniques and show that they can be utilized in a software context.

- To demonstrate that the software clustering field has research potential.

The remainder of this dissertation presents our approach to addressing several of the open questions related to the field of software clustering. We begin by tackling the problem of comparing the results of different software clustering techniques.

Chapter 3

Comparison of clustering techniques

3.1 Motivation

As evidenced in the previous chapter, the software clustering problem has attracted the attention of many researchers. This has resulted in a variety of approaches being published. However, the software clustering literature contains a surprisingly small amount of work on comparing existing software clustering techniques.

There are many reasons for this situation. These include:

1. Many of the approaches presented in the literature are not accompanied by an implementation, or are only available as research prototypes that tend to be rather user-unfriendly.

2. Most software clustering techniques are validated by applying them to example systems that are either proprietary or too small to draw any general conclusions from. There exists a need for software systems of considerable size, well-known

structure, and ready availability, that can be used as benchmarks. Popular open-source systems such as Netscape or Linux would be good candidates.

3. Even if it becomes possible to test several techniques on the same large system, it is not clear how to evaluate the results. The evaluation criterion used by most methods is developer feedback, in which system designers and seasoned developers offer their opinion on how well the obtained clustering reflects the actual breakdown of their system. This method however incurs a large degree of subjectivity, and it becomes impractical when several techniques are being tested.

From the above, it becomes clear that an objective means of measuring the effectiveness of a clustering approach is required.

We present a metric called MoJo that gauges how "close" two different clusterings of the same set of objects are. In other words, the metric accepts two distinct decompositions of the resources of a software system as input, and outputs a number that represents a measure of the "distance" between them. Such a metric will be helpful in our effort to compare the effectiveness of different clustering algorithms in the following ways:

1. Many of the approaches presented in the literature are parameterized, i.e. they require the tuning of the value of certain parameters in order to be effective [MMR+98, MOTU93, JD88]. Using our metric we can determine how drastic an effect a change of value has on the derived clustering, and thus assess the importance of each parameter.

2. An important feature of clustering algorithms that hasn't attracted as much attention as it should so far, is their stability, i.e. how is the output of the algorithm

(the obtained partition) affected when its input (the software system) is slightly modified. Researching this issue can be aided by our metric. It allows us to measure the effect small changes in the software system have on the output of a clustering algorithm.

3. In the cases of software systems for which an "authoritative" clustering already exists (as is the case with systems that have been clustered manually by their architect), this metric will help us determine how close a given algorithm can come to reproducing it.

The next section describes the types of decompositions we consider. In section 3.3, we present the model we use in order to measure similarity between two partitions. Section 3.4 describes a heuristic algorithm that approximates the value of the metric efficiently. Justification for the choices made by the heuristic algorithm is discussed in section 3.5, while section 3.6 presents some experiments that we have performed using an implementation of the heuristic algorithm. These experiments showcase the performance of the algorithm and the effectiveness of the metric.

3.2 Flat vs. nested decompositions

It is important to note that for the purposes of this chapter, we are dealing only with flat decompositions. In other words, the clusterings we consider do not contain nested clusters. There is only one level of clusters and one level of objects (figure 3.1 presents the containment tree of an example flat decomposition).

This restriction, however, does not limit the applicability of our metric. Any non-flat

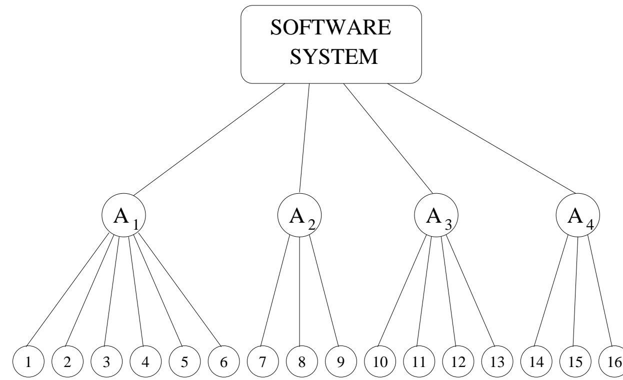


Figure 3.1: The containment tree of a flat decomposition. The nodes labeled A_i are clusters, while the leaves of the tree are the objects.

(also called *nested*) decomposition can be transformed to a flat one with minimal loss of information. This can be done in two different ways depending on whether we require a compact or a detailed flat decomposition:

1. Converting a nested decomposition to a compact flat one. Each object is assigned to its ancestor that is closest to the root of the containment tree, i.e. to its ancestor of height 2. The flat decomposition obtained contains only the top-level clusters of the original one (figure 3.2 presents an example of such a conversion).

2. Converting a nested decomposition to a detailed flat one. Each cluster is assigned directly to the root of the containment tree, i.e. each cluster and its sub-tree are moved so that the height of the cluster becomes 2. The decomposition obtained contains any cluster that contained at least one object in the original decomposition (an example is presented in figure 3.3).

In the next section, we define our metric and discuss the rationale behind our definition.

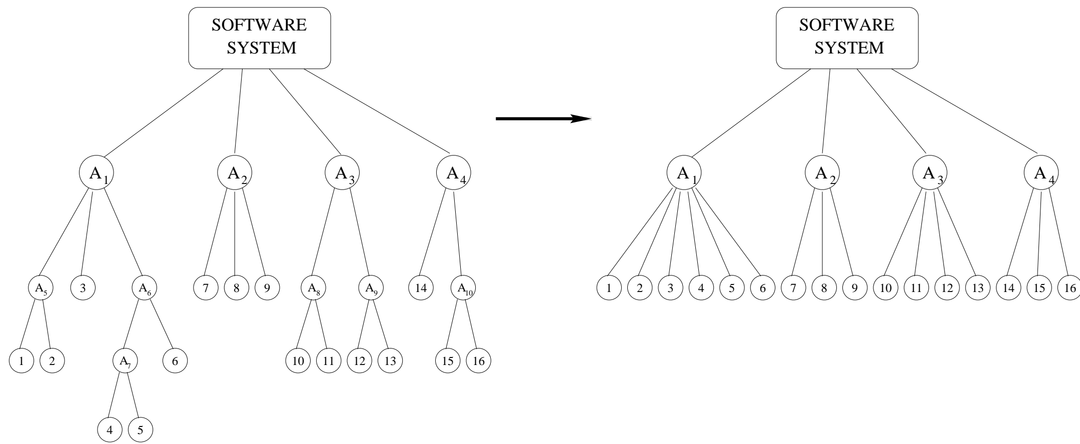


Figure 3.2: Conversion of a nested decomposition to a compact flat decomposition.

3.3 The MoJo Distance Metric

In order to compute the distance between two partitions of the same set of objects, MoJo uses an idea that has been used in other disciplines that employ clustering techniques[RY81]: it counts the minimum number of operations (such as moving a resource from one cluster to another, joining two clusters etc.) one needs to perform in order to transform one partition to the other.

MoJo allows only two operations (it is from these two operations that MoJo gets its name):

1. *Move* : **Mo**ving an object from one cluster to another. This includes moving a resource into a previously non-existent cluster, thus creating a new cluster of cardinality 1.

2. *Join* : **Jo**ining two clusters into one, thus reducing the number of clusters by 1.

Let us denote by $mno(A, B)$ the minimum number of operations to transform partition

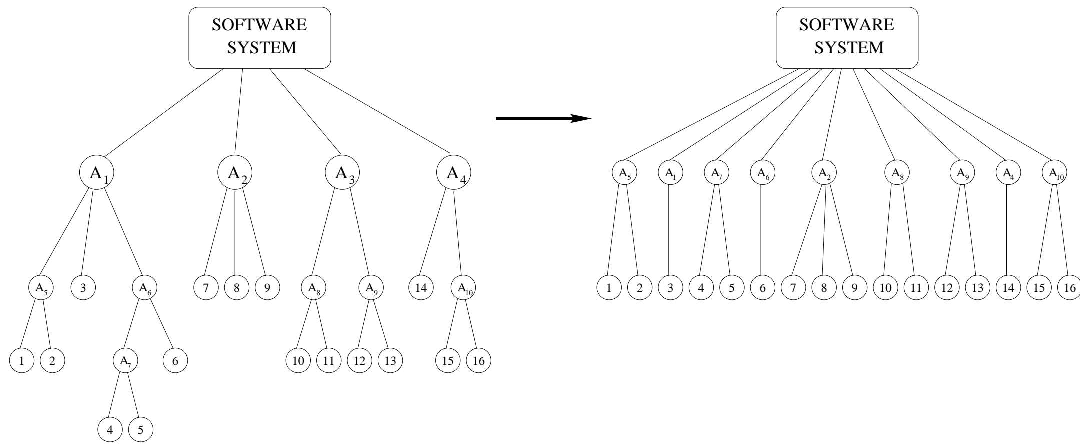


Figure 3.3: Conversion of a nested decomposition to a detailed flat one.

A into partition B. The value of our metric[1] is defined as:

$$MoJo(A, B) = min(mno(A, B), mno(B, A))$$

It is interesting to note that it does not necessarily hold that $mno(A, B) = mno(B, A)$. For example, partition A in figure 3.4 can be transformed to partition B with a single *Join* operation. Therefore, we have $mno(A, B) = 1$. However, $mno(B, A) = 3$ because three resources must be moved to split B's single cluster.

The definition of the MoJo metric warrants some further discussion. A reasonable criticism would be the following: Why is the *Split* operation disallowed? Figure 3.5 provides some intuition for this decision. It presents three different clusterings of the same set of 20 objects. Using clustering C_0 as a reference, we can see that clustering C_1 is much closer to it than clustering C_2. However, if a *Split* operation was allowed, we would have $MoJo(C_1, C_0) = MoJo(C_2, C_0) = 2$ since both transformations could be

[1]The term *metric* is not used in its strict mathematical sense here. For instance, it can be shown that the MoJo metric does not satisfy the triangle inequality.

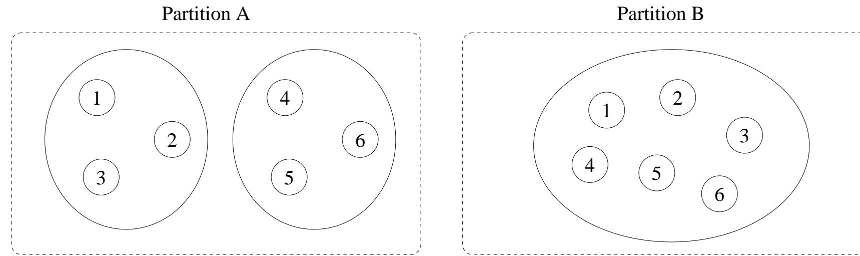


Figure 3.4: Two partitions that demonstrate the non-symmetrical behaviour of the *mno* function.

accomplished by joining the two clusters of each partition and then splitting accordingly.

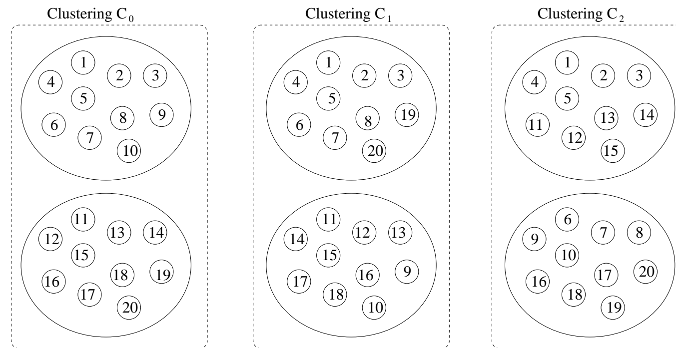


Figure 3.5: Three partitions that demonstrate the reason for the disallowance of the *Split* operation.

Since a *Split* operation can always be simulated by a series of *Move* operations, one can view the MoJo model from a different point of view. The *Split* operation is not really disallowed, it is just assigned a larger weight than *Move* and *Join*. That weight is equal to the cardinality of the smaller of the two new clusters.

Other interesting features of the MoJo metric are the fact that we chose to disallow the *Split* operation as opposed to the *Join* one, as well as the assignment of equal weight to both *Move* and *Join* (each carries a weight of 1). There were several reasons for these

choices:

- From a mathematical point of view, *Move* and *Join* are operations that can only be done in one way (as opposed to the operation of splitting a cluster of size n into two, which can be done in $2^{n-1} - 1$ ways).
- From a software clustering point of view, the fact that we need to join two clusters means that we performed more detailed clustering than required, hardly a big problem. Having to split a cluster, however, indicates that our clustering was rather crude.
- From an information theory point of view, there is always a constant amount of information one needs to provide in order to distinctly identify a *Move* or a *Join* operation (the name of the object to be moved, and the names of the clusters involved are sufficient). However, a *Split* operation requires a list of all objects to be placed in each cluster, something that can vary depending on the size of the cluster in question.

Finally, the standard Information Retrieval metrics of Precision and Recall have also been used in the software clustering literature in order to compare clusterings [AL99]. It is done by considering pairs of entities. Two entities are either in the same cluster (this is called an "intra" pair) or in two different clusters (an "inter" pair). By considering one of the partitions as the "reference" partition, precision and recall for the other partition (we will call it the "proposed" partition) are computed as follows:

- *Precision*: Percentage of intra pairs in the proposed partition which are also intra in the reference partition.

- *Recall*: Percentage of intra pairs in the reference partition which are also found in the proposed partition.

There is no doubt that these metrics can also provide a measure of the similarity between two partitions. However, there were a number of reasons why we did not adopt them:

- In many cases, there is no reference partition. One can overcome this problem by arbitrarily assigning one of the partitions as the "reference" one. However, in that case the metrics become interchangeable (the value of the precision metric when one partition is chosen as the "reference" partition is equal to the value of the recall metric when the other partition is chosen as "reference" and vice versa). This makes comparing different results difficult, especially in the case where the value of one metric increases while the value of the other decreases.

- The fact that one needs two numbers to express similarity between clusterings was not suitable for our intended use of this metric (examples include the cluster quality metric in section 3.6.3 and the stability experiments in chapter 4).

In the next section, we present a heuristic algorithm that approximates the value of our metric in an efficient manner.

3.4 The heuristic algorithm

Finding the exact value of $MoJo(A, B)$, where A and B are two partitions of the same set of objects S, can be time-consuming when S is large. It has been shown [ZSS92] that

finding such a distance when the operations used are *Insert* and *Delete* (instead of *Move* and *Join*) is an NP-hard problem.

We have implemented a heuristic algorithm that computes an approximation to the value of $MoJo(A, B)$ in a reasonable amount of time (the name of our heuristic algorithm is HAM for Heuristic Algorithm for MoJo). Experiments using this implementation are presented in section 3.6. In this section we present the heuristic algorithm used.

Our algorithm computes an approximation to $mno(A, B)$. From that, we can compute an approximation to $MoJo(A, B)$ as shown in the previous section. The computation is done by actually transforming partition A to B while keeping track of the number of operations we perform. We will denote the number of operations that have been performed at any stage by $HAM(A, B)$. To start things off, $HAM(A, B)$ is set to 0.

Let $A_i, 1 \leq i \leq l$ be the clusters of partition A (an example with 25 objects is shown in figure 3.6), and $B_j, 1 \leq j \leq m$ be the clusters of partition B (a corresponding example is shown in figure 3.7). Throughout the presentation of the algorithm, we will use this example to illustrate the transformations that have been performed on partition A.

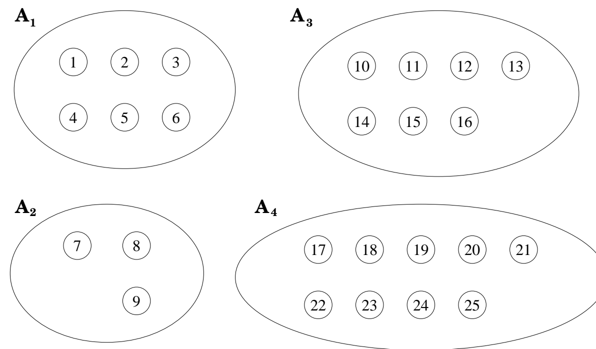


Figure 3.6: Example partition A comprising 25 objects.

HAM goes through a number of stages in order to accomplish its task. We will present

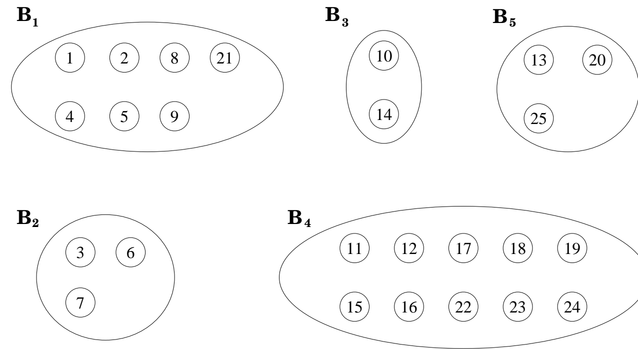


Figure 3.7: Example partition B comprising the same 25 objects as partition A.

them in the following in order of execution.

3.4.1 Tag assignment

Each object s in partition A is assigned a tag $T_j, 1 \leq j \leq m$ depending on which cluster s belongs to in partition B. For example, object ① is assigned tag T_1 since it belongs in cluster B_1, and object ③ is assigned tag T_2 since it belongs in B_2. The tags are shown in figure 3.8 as a rectangle next to each object (only the subscripts are shown to avoid cluttering, i.e. tag T_2 is shown as [2]).

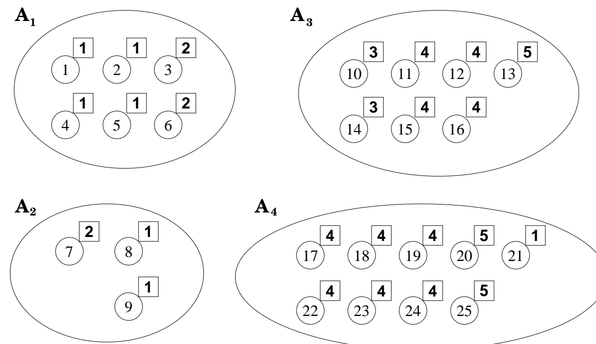


Figure 3.8: Partition A with the tags.

Let us denote by $A_i(T_j)$ the set of objects that are contained in cluster A_i and are assigned the tag T_j. For example, $A_1(T_1)$ contains objects ①, ②, ④, and ⑤, while $A_4(T_5)$ contains objects ⑳ and ㉕. The cardinality of set $A_i(T_j)$ denotes the amount of overlap between clusters A_i and B_j. To simplify things, we define v_{ij} to be $|A_i(T_j)|$.

Our algorithm considers only the tags of each object from now on. Figure 3.9 presents partition A including only the tags of each object. At the end of this stage, $HAM(A, B)$ is still 0.

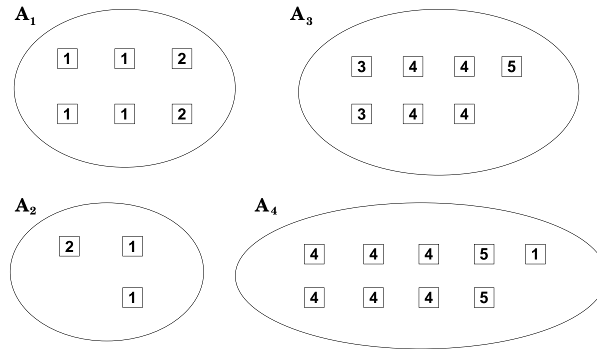


Figure 3.9: Partition A containing only tags.

3.4.2 Partner assignment

We now introduce the notion of *partner* cluster. Each cluster B_j in partition B is assigned a partner cluster A_i in partition A, such that $v_{ij} \geq v_{kj}, 1 \leq k \leq l$. If more than one of the clusters of A fit this definition, one of them is chosen arbitrarily. We will denote the partner cluster of cluster B_j by P_j. For example, we have $P_4 = A_4$, since cluster A_4 contains the largest number of objects tagged with T_4.

The intuition behind partner clusters is that during our effort to transform A to B we will attempt to transform P_j (which is a cluster in partition A) to B_j (a cluster in

partition B). Figure 3.10 presents partition A after partner clusters have been assigned. $HAM(A, B)$ is still 0.

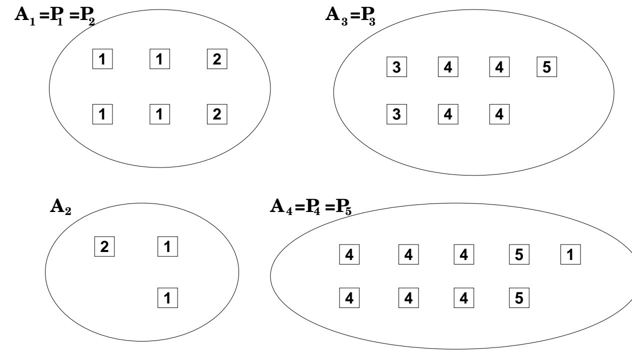


Figure 3.10: Partition A after partner clusters have been assigned.

3.4.3 Partner refining

The aforementioned definition of partner clusters can result in the same cluster A_i being the partner cluster of more than one clusters in B (in our example, we have $A_1 = P_1 = P_2$, since cluster A_1 contains the largest subsets of $\boxed{1}$s and $\boxed{2}$s). However, this would pose a problem when the time would come to transform clusters P_i to their corresponding clusters B_i. As a result, we refine the assignment of partner clusters by repeating the following procedure (shown as indented text) until all partner clusters are distinct or no further improvement can be made. The intuition behind it is that, if we have $A_z = P_x = P_y$, we try to find the second largest subsets of $\boxed{x}$s and $\boxed{y}$s (in our example, A_2 contains the second largest subsets of $\boxed{1}$s and $\boxed{2}$s, so this procedure will attempt to make A_2 the partner cluster of either B_1 or B_2).

Let us assume that we have integers x, y, and z, such that $1 \leq x, y \leq m$, $1 \leq z \leq l$, and $A_z = P_x = P_y$ (in our example we would have $x = 1$, $y = 2$,

$z = 1$). We also define sets $\Phi_k, 1 \leq k \leq m$ that are initially empty (these sets are used as markers, so that we do not consider the same A_i more than once). We compute the following values:

$$v_{fx}, \text{ such that } v_{fx} \geq v_{ix}, 1 \leq i \leq l, i \neq z, i \notin \Phi_x$$

$$v_{gy}, \text{ such that } v_{gy} \geq v_{jy}, 1 \leq j \leq l, j \neq z, i \notin \Phi_y$$

If both $v_{fx} = 0$ and $v_{gy} = 0$, then no other A_i can be considered to be the partner cluster of either B_x or B_y, so no change is made.

If only $v_{fx} = 0$, then we assign $P_y = A_g$ and add z to Φ_y.

If only $v_{gy} = 0$, then we assign $P_x = A_f$ and add z to Φ_x.

Otherwise, if $v_{zx} + v_{gy} > v_{fx} + v_{zy}$, then we assign $P_y = A_g$ and add z to Φ_y, else we assign $P_x = A_f$ and add z to Φ_x.

The effect of this process is that if there existed a cluster A_i that was the partner cluster of two clusters from partition B, an attempt is made to assign a different cluster of A as the partner cluster of one of them.

In our example, we have $v_{11} = 4$, $v_{12} = 2$, $v_{21} = 2$, and $v_{22} = 1$, while $A_f = A_2$ and $A_g = A_2$. According to the above, we will now assign $P_2 = A_2$. Also, since we have $A_4 = P_4 = P_5$, this process will also assign A_3 as the new P_5. However, A_3 is also P_3, and since no other cluster contains any objects tagged with either T_3 or T_5, we will have $A_3 = P_3 = P_5$ at the end of this stage. Figure 3.11 shows partition A at this point. No *Move* or *Join* operations have been performed so far, so $HAM(A, B)$ remains 0.

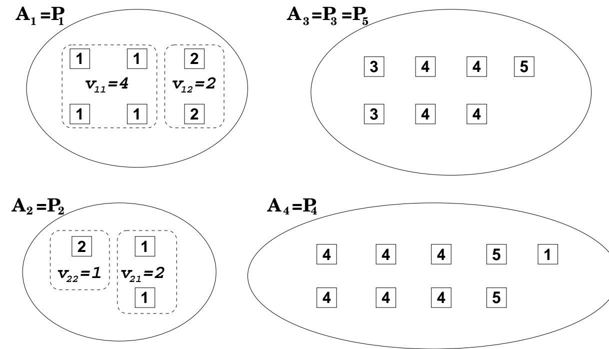


Figure 3.11: Partition A after the partner refining stage.

3.4.4 Partner splitting

As demonstrated by our example, it is still possible that there exist non-distinct partner clusters after the end of the previous stage (this will surely happen if $|A| < |B|$). In this case, our algorithm begins the transformation by "splitting" these clusters into two, so that one of the two new clusters contains only the objects tagged with T_x, where $v_{zx} \leq v_{zy}$. This splitting is simulated as a series of *Move* operations, and the algorithm adds their number to the current value of $HAM(A, B)$. When this stage is over, every cluster B_j should have a distinct partner cluster in partition A.

In our example, we have $A_3 = P_3 = P_5$ and $v_{33} = 2$ while $v_{35} = 1$. As a result, the single object tagged with T_5 is moved into its own cluster called simply P_5. Figure 3.12 shows partition A at the end of this stage. The value of $HAM(A, B)$ is now 1.

3.4.5 Profitable joins

At this point, a simple series of operations that accomplishes the transformation of partition A to partition B is one that moves all objects tagged with T_j into P_j. However, in

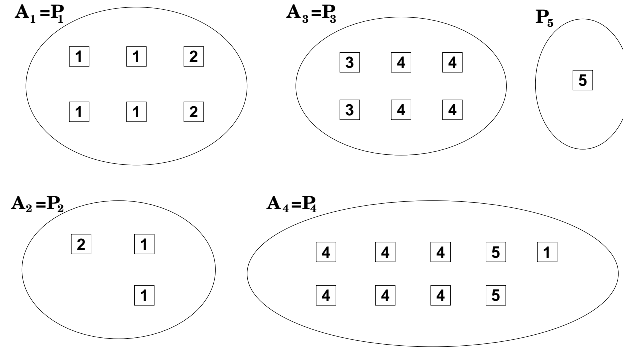


Figure 3.12: Partition A containing distinct partner clusters.

this way we do not benefit from the fact that joining two clusters has a weight of only 1. As a result, our algorithm will attempt to perform any "profitable" *Join* operations before resorting to *Move* operations.

In order to decide what a profitable *Join* operation is, let us consider two clusters $\mathbb{X} = P_x$ and $\mathbb{Y}$. If $\mathbb{Y}$ is not a partner cluster, then it is profitable to join the two clusters if $|\mathbb{Y}(T_x)| > 1$. If $\mathbb{Y}$ is a partner cluster, such that $\mathbb{Y} = P_y$, then we proceed as follows:

Let us denote by w_{ij} the overlap between clusters P_i and B_j. More formally, we define $w_{ij} = |P_i(T_j)|$. Let us also assume without loss of generality that $w_{yx} \geq w_{xy}$ (if not, we can interchange clusters $\mathbb{X}$ and $\mathbb{Y}$). Transforming P_x to B_x and P_y to B_y by using only *Move* operations will require a cost of $|P_x| - w_{xx} + |P_y| - w_{yy}$. At the same time, if we first join these two clusters into a new cluster $\mathbb{Z}$, and then proceed to transform to B_x and B_y (by moving everything except for $\mathbb{Z}(T_x)$ out of $\mathbb{Z}$, and creating a new cluster for $\mathbb{Z}(T_y)$), our cost will be $1+|P_x|-w_{xx}+|P_y|-w_{yx}$. Therefore, such a *Join* operation is profitable only if $|P_x|-w_{xx}+|P_y|-w_{yy} < 1+|P_x|-w_{xx}+|P_y|-w_{yx} \Rightarrow 1-w_{yx} < -w_{yy} \Rightarrow w_{yx} > w_{yy}+1$.

As a result, by looking at each cluster P_i, we can decide if it is profitable to join it with any cluster P_j, by determining if the inequality $w_{ij} > w_{ii} + 1$ holds. As figure 3.13

illustrates, this is true in our example with clusters P_3 and P_4.

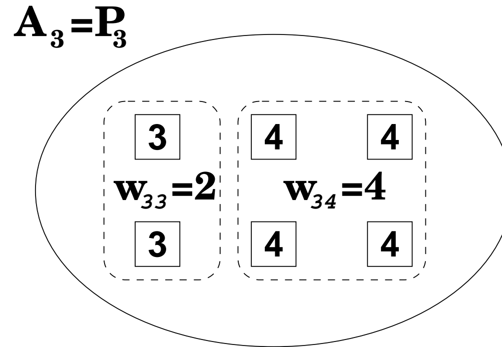


Figure 3.13: Even though cluster A_3 is P_3, the cluster contains more objects tagged with T_4 than ones tagged with T_3 ($w_{34} > w_{33} + 1$). This indicates that it is profitable to join clusters P_3 and P_4.

Thus, our algorithm determines if such profitable *Join* operations exist by examining all clusters. It then performs the *Join* operation that will save the largest number of operations (the number of operations saved by a profitable *Join* operation is $w_{ij} - w_{ii} - 1$). All clusters are then examined again, and this process repeats until no further profitable *Join* operations exist.

In our example, it is profitable to join clusters A_3 and A_4. Therefore, HAM will join these two clusters into a new cluster we will call A_{34}. The value of $HAM(A, B)$ will also increase by 1. The objects tagged with T_3 will now have to be moved out of A_{34} into a new cluster called P_3. This will further increase $HAM(A, B)$ by 2, for a total of 4 at the end of this stage. Figure 3.14 presents partition A after clusters A_3 and A_4 have been joined. Since no further profitable *Join* operations exist, this will also be the shape of partition A at the end of this stage.

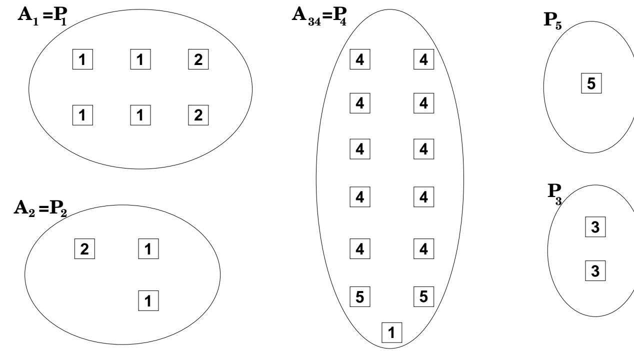


Figure 3.14: Partition A after all profitable *Join* operations have been performed.

3.4.6 Move operations

Finally, when no more profitable *Join* operations remain, our algorithm moves all objects tagged with T_j into P_j. In our example, there are 7 such *Move* operations to be performed. As a result, the final value of $HAM(A, B)$ will be 11 (this happens to be the exact value of $mno(A, B)$ as well).

Figure 3.15 presents partition A at the end of this last stage. It is easy to see that it is isomorphic to partition B shown in figure 3.7.

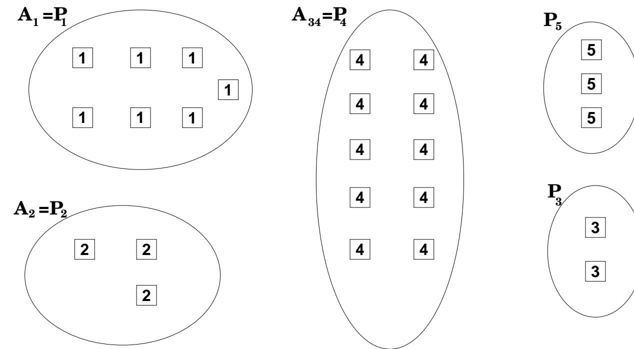


Figure 3.15: Partition A at the end of the execution of the HAM algorithm.

The next section presents some insight on why we believe that the algorithm presented

above provides a good approximation to the actual MoJo value.

3.5 Algorithm analysis

The heuristic algorithm described in the previous section makes some assumptions in order to compute the MoJo distance. In this section, we will show that these assumptions have little or no effect on the value computed. This should suggest that our algorithm computes a close approximation of the actual value, something corroborated by the experiments presented in section 3.6.

There are three assumptions that our algorithm makes that could compromise its accuracy:

1. Commutativity of *Move* and *Join* operations.
2. Order of performing *Join* operations.
3. Partner cluster selection.

We will examine each one closely in the following.

3.5.1 Commutativity of *Move* and *Join* operations

Our algorithm performs all profitable *Join* operations first, and then concludes with a series of *Move* operations (some *Move* operations may be performed after each *Join* operation). We will show that there exists a minimum-length sequence of operations of this form, by showing that any two operations in a minimum-length sequence can be swapped.

In the following, we will denote by $\mathbb{M}_{A\to B}$ a *Move* operation that moves an object from cluster A to cluster B. Also, $\mathbb{J}_{A+B}$ will denote a *Join* operation that merges clusters A and B.

Lemma 1 *Any two Move and Join operations can be swapped.*

There exist three distinct relevant possibilities for an $\mathbb{MJ}$ sequence (note that the $\mathbb{M}_{A\to B}\mathbb{J}_{A+B}$ sequence is not possible since it is equivalent to $\mathbb{J}_{A+B}$). All three cases are listed below with their equivalent $\mathbb{JM}$ sequences.

$$\begin{aligned}
\mathbb{M}_{A\to B}\mathbb{J}_{C+D} &\equiv \mathbb{J}_{C+D}\mathbb{M}_{A\to B} \\
\mathbb{M}_{A\to B}\mathbb{J}_{B+C} &\equiv \mathbb{J}_{B+C}\mathbb{M}_{A\to BC} \\
\mathbb{M}_{A\to B}\mathbb{J}_{A+C} &\equiv \mathbb{J}_{A+C}\mathbb{M}_{AC\to B}
\end{aligned}$$

Lemma 2 *Any two Join operations can be swapped.*

This is trivial to see when the two *Join* operations refer to different clusters. If they refer to the same clusters (as in $\mathbb{J}_{A+B}\mathbb{J}_{AB+C}$) the associativity of the *Join* operation allows us to perform these operations in any order.

Lemma 3 *Any two Move operations can be swapped.*

This is obvious when the two *Move* operations refer to different objects. However, two subsequent *Move* operations in a minimum-length sequence cannot refer to the same object.

From the above lemmas, we can deduce that any two operations in a minimum-length sequence can be swapped. This shows that the fact that HAM produces a sequence of operations of a specific form does not constitute a limitation.

3.5.2 Order of performing *Join* operations

Of all the profitable *Join* operations that may exist at any point, our algorithm chooses one that will save the largest number of operations. The two selected clusters are joined, and the process is then repeated with the obtained clustering. Even though such a greedy approach can suffer from a local optima problem, it has been shown to work accurately on many small-sized examples.

Figure 3.16 presents an example with three clusters. An arrow from a cluster P_i to a cluster P_j indicates that it is profitable to join these clusters, and that the newly formed cluster will be P_j (P_i will have to be reformed by moving all the objects tagged with T_i out of P_j).

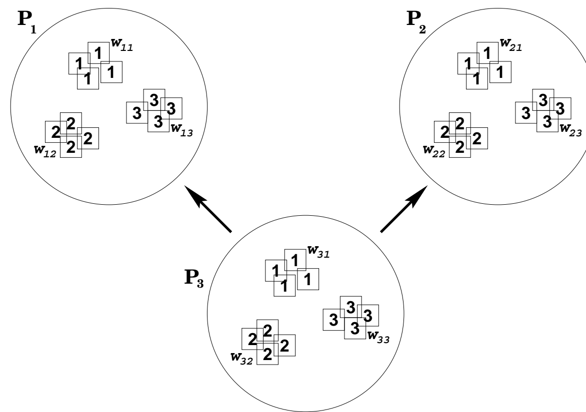


Figure 3.16: An example that demonstrates the efficiency of the order of *Join* operations that HAM selects.

Depending on whether there is an arrow or not between any pair of clusters in our example, we know that the following inequalities hold:

$$w_{12} \leq w_{11} + 1 \qquad\qquad w_{13} \leq w_{11} + 1$$

$$w_{21} \leq w_{22} + 1 \qquad w_{23} \leq w_{22} + 1$$

$$w_{32} > w_{33} + 1 \qquad w_{31} > w_{33} + 1$$

Let us assume without loss of generality that $w_{32} > w_{31}$. This would mean that HAM would choose to join clusters P_2 and P_3 first. The total number of operations that this would save is $w_{32} - w_{33} - 1$ (after P_2 and P_3 are joined, there will exist no more profitable *Join* operations).

We will now show that had we chosen a different order of *Join* operations there would be no gain. If we join clusters P_1 and P_3 first, we save $w_{31} - w_{33} - 1$ operations. In order to do better than the alternate order of operations, we have to assume that it will be profitable to join the new P_1 and P_2 next. That will save an additional $w_{12} + w_{32} - w_{11} - w_{31} - 1$. The total number of operations that we will save will be $w_{12} + w_{32} - w_{11} - w_{33} - 2$.

Therefore, the number of extra operations we might save using the alternate ordering is $w_{12} - w_{11} - 1$. However, this is a non-positive quantity. As a result, the alternate ordering is at best just as good as the one HAM picks, and will usually be worse.

Clusterings of real software systems are unlikely to contain many profitable *Join* operations, making the impact of the order of performing them minimal.

3.5.3 Partner cluster selection

An integral part of the heuristic algorithm is the selection of the partner clusters. The better the assignment is, the fewer operations our algorithm will have to perform. The way our algorithm selects the partner clusters is indeed geared towards the minimization of the number of subsequent operations.

By assigning as partner cluster of a cluster B_j, the cluster A_i that contains the largest

subset of objects tagged with T_j, our algorithm ensures that the number of *Move* operations that will have to be performed to transform A_i into B_j is minimized. Furthermore, the procedure that ensures that there is a distinct partner cluster of each B_j reassigns partner clusters in a way that minimizes the number of further operations as well.

In the next section, we present some experiments conducted using an implementation of our algorithm.

3.6 The experiments

The algorithm described in section 3.4 has been implemented. The implementation has been used to conduct several experiments. In this section we present some of these experiments that showcase the algorithm's efficiency as well as the usefulness of a metric such as MoJo.

3.6.1 Performance evaluation

The first experiment demonstrates that the value of the metric can be computed in a reasonable amount of time and that its computational complexity is no worse than linear with respect to the number of objects contained in each partition. We utilized two sources of data:

1. *Random data.* We have devised a clustering generator that, given an integer n creates a clustering of as many objects (the number of clusters is a random number between 5 and $\lfloor \frac{n}{5} \rfloor$). Our algorithm is then used to compute the distance between two such clusterings.

2. *Field data.* Our research team is associated with two large software systems. These systems have been clustered manually by domain experts. We have also attempted to cluster these two systems with the ACDC algorithm that will be presented in chapter 6. We used HAM to compute the distance between the manual clustering and the one obtained by running ACDC.

Type of data	Objects	Time
Random	200	0.17 sec
Random	600	0.7 sec
Field	939	0.67 sec
Random	1000	3.38 sec
Random	1400	4.85 sec
Random	1800	6.79 sec
Field	2097	6.93 sec
Random	2200	8.84 sec

Table 3.1: Summary of results.

Table 3.1 summarizes our findings. Note that the values presented for the random data are average values obtained after creating 20 different clusterings and computing the distance between each pair (190 comparisons).

Figure 3.17 depicts a plot of our experimental results. We have also fit these data to a linear function via the least squares method. The fitted curve, that can also be seen in the same figure, seems to fit the experimental data fairly well. Only small object numbers (such as less than 500 or so) seem to deviate from this behaviour. However,

this is an expected phenomenon since there exist performance overheads that would only matter if the number of objects is small. Besides, the run-time for these experiments was less than one second; hardly a cause for concern.

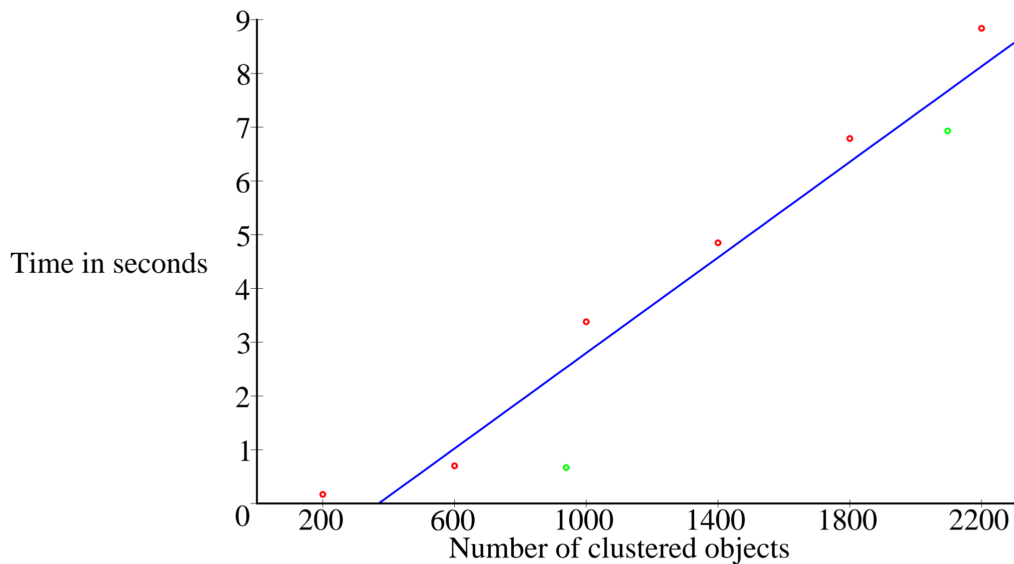


Figure 3.17: A graphic depiction of the performance experiment results. Field data are shown as green dots, while random data are red.

An interesting observation is that HAM seems to perform better with field data rather than random data. Random data points are almost always to the left of the fitted curve, while field data points are always on its right. One of the reasons is that field data have similarities in their decompositions, something that our heuristic exploits in order to run faster.

3.6.2 Accuracy evaluation

In order to measure our algorithm's accuracy, we implemented a program that performs a series of random *Move* and *Split* operations to an initial clustering. Our intention was to compare the value of the distance between the initial and the altered clustering that our algorithm calculates to the number of operations we performed. It should be noted that the number of *Move* and *Split* operations we performed is only an upper bound of the actual MoJo value since there is no way to guarantee that the sequence of *Move* and *Split* operations is minimal. However, by repeating the experiment many times one can get a good idea of our algorithm's accuracy.

For this experiment we used a random partition of 2200 objects into 106 clusters. This partition was randomly modified by performing k operations on it, where k took all the values between 1 and 100. This process was repeated 1000 times for a total of 100000 different experiments.

The results were pleasantly surprising. In 83% of the experiments our algorithm found the exact value of operations we performed. In the remaining 17% it found an even smaller value. The fact that our algorithm was never off indicates that it is performing well, but it might suggest that more thorough testing has to performed.

3.6.3 Clustering quality metric

A metric like MoJo can also be used in order to evaluate the effectiveness of clustering algorithms. For this to happen, we need to have an "authoritative" clustering B of a software system S, i.e. a clustering that has been approved by someone knowledgeable about the system, such as the system designers or seasoned developers. We can then

evaluate how close a clustering algorithm comes to reproducing the authoritative clustering B by computing the number of operations one needs to perform in order to transform the algorithm's output (we will call it clustering A) to B.

We propose the following quality metric for a software clustering algorithm (SCA) that has been applied to a software system S (n is the number of objects clustered):

$$Q(SCA, S) = \left(1 - \frac{mno(A, B)}{n}\right) \times 100\%$$

It holds that $mno(A, B) < n$ (any partition of n objects can be transformed into any partition of the same set of objects by performing $n - 1$ *Move* operations). Therefore, $Q(SCA, S)$ will always have a value between 0% and 100% (100% is only achieved if $A = B$, i.e. the clustering algorithm produces the same partition as the system designers).

In the above formula, we have used $mno(A, B)$ instead of $MoJo(A, B)$. The reason for this is the fact that we have a "reference" clustering (the authoritative clustering B), so we are only interested in how close A comes to B (expressed by $mno(A, B)$), and not in how close B comes to A (expressed by $mno(B, A)$ and possibly by $MoJo(A, B)$). Also, using $mno(A, B)$ instead of $MoJo(A, B)$ avoids the following paradox:

Let ONE be a clustering algorithm that always produces a clustering of cardinality 1 (all objects in one cluster). The quality of such an algorithm should be very low. However, if we use $MoJo(A, B)$ in our quality measure, the computed value will be very high. Using TOBEY[2] as an example (its authoritative clustering contains 69 clusters and 939 objects) we would have $Q(ONE, TOBEY) = (1 - 68/939) \times 100\% = 92.8\%$, a value no "normal" clustering algorithm could ever hope to reach. This is because the authoritative clustering of TOBEY can be transformed into the output of algorithm ONE

[2]TOBEY is a large industrial software system. See Appendix A for a detailed description.

by just joining all its clusters. Using $mno(A,B)$ instead, we get the more reasonable value of 17.9% for the quality of algorithm ONE.

Interestingly enough, the quality metric value of algorithm EACH, which always produces a clustering where each cluster contains exactly one object, is the same whether we use $mno(A,B)$ or $MoJo(A,B)$ in our formula. In the case of TOBEY, this value is 7.3%.

To provide a more meaningful example, we computed the quality metric value of Bunch[3] using both the NAHC and SAHC algorithms (TOBEY was used as our input). The results were:

$$Q(NAHC, TOBEY) = \left(1 - \frac{431}{939}\right) \times 100\% = 54.1\%$$

$$Q(SAHC, TOBEY) = \left(1 - \frac{486}{939}\right) \times 100\% = 48.2\%$$

These values denote that Bunch can be of help to reverse engineering projects since it appears to be able to correctly cluster a large portion of the software system at hand. As a means of comparison, we computed the quality of our random clustering generator. The best value obtained after creating 10000 partitions was 23.4%, while the average value was approximately 12.8%.

This quality metric can also be used to evaluate future versions of a clustering algorithm in order to decide if they constitute an improvement.

3.6.4 Parameter tuning

Many approaches presented in the software clustering literature are parameterized, i.e. they require appropriate choices for the values of certain parameters in order to maximize

[3] Bunch is a public domain clustering tool. See appendix B for a detailed description.

their effectiveness. MoJo can aid the process of choosing these values by means of the aforementioned quality metric.

For this experiment we used Bunch, and we experimented with its genetic algorithm part. Certain parameters such as the number of generations, the population size, and the crossover probability require choice of value (default values are provided but they may not be the optimal ones). The following table presents experiments with various values of these parameters and the corresponding value of the quality metric (TOBEY was used as input). From these results, one can deduce that a larger population size and crossover probability, together with a smaller number of generations seem to be the appropriate choices.

Number of generations	Population size	Crossover probability	Q(Bunch,TOBEY)
500	50	0.5	48.35%
500	50	1	49.41%
500	100	0.5	47.39%
500	100	1	52.08%
1000	50	0.5	43.88%
1000	50	1	40.04%
1000	100	0.5	41.32%
1000	100	1	43.45%

Table 3.2: Quality metric values for various combinations of values for 3 parameters.

Such experimentation can help the user of a parameterized tool maximize its perfor-

mance.

This chapter presented MoJo, a metric that can help the process of comparing and evaluating software clustering approaches. We also presented a heuristic algorithm for the effective calculation of its value.

In the next chapter, we will utilize MoJo in order to assess the stability of various software clustering algorithms.

Chapter 4

Stability of clustering algorithms

4.1 Introduction

One of the biggest challenges a software clustering algorithm (SCA) faces is that of surviving as part of the daily implementation cycle of a real-life software development project. In other words, a good way to measure the effectiveness of an SCA would be to include it in the daily routine of a developing team. Assuming that the system is built nightly, the SCA would produce a new decomposition of the software system that would be shown to the developers the next morning, probably in the form of diagrams produced by a visualization tool.

For an SCA to be accepted as part of the daily routine, it would have to produce "meaningful" decompositions, something all SCAs strive to do. However, another factor might also play an important role. The SCA will have to adhere to the "rule of minimal change", i.e. the diagrams presented each morning should not be radically different from the ones shown the previous day, since that could lead to confusion, and eventually the

rejection of the SCA.

This raises an issue that has not attracted much attention from the software clustering community so far: an SCA's *stability*. It would be interesting to know how much is the output of an SCA affected, when its input is slightly modified. It is a desirable feature for an SCA to be stable.

This chapter focuses on the issue of stability. We present a measure for it, and attempt to demonstrate its importance. Section 4.2 presents a definition of the stability problem, and discusses various facets of the notion of stability. Section 4.3 introduces a measure for the stability of a given SCA. Section 4.4 describes experiments we performed with this measure and the obtained results. Finally, Section 4.5 concludes the chapter.

4.2 Definitions

The notion of stability has been defined and studied in a variety of contexts, such as control systems, sorting algorithms, and even social sciences. In the context of a software clustering algorithm, we define an SCA to be stable, if the obtained clusters are not grossly affected by slight modifications in its input, i.e. the software system in question.

More precisely, let S_0 be the software system, and A the algorithm we are interested in. When we apply A on S_0 we obtain a clustering C_0. Subsequently, S_0 is modified slightly. Let us call the new system $S_1 = S_0 + \Delta S_1$. Applying the clustering algorithm on S_1, we obtain a new clustering $C_1 = C_0 + \Delta C_1$ (see figure 4.1). If ΔC_1 is consistently small when ΔS_1 is small, then A is deemed to be *stable* with respect to S_0. By applying algorithm A to a variety of software systems, we can determine whether it behaves in a stable fashion.

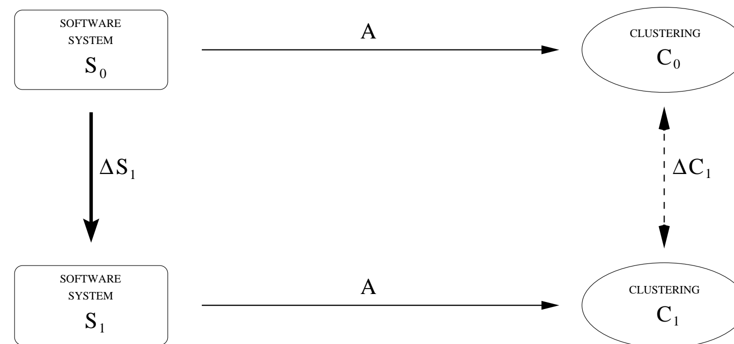


Figure 4.1: When a software system is slightly modified, a software clustering algorithm should produce similar results.

The aforementioned definition of stability raises two important questions:

1. What constitutes a "slight modification" of a software system? What extent of changes to a software system is too large to be considered "slight" anymore, i.e. how big of a ΔS_1 do we consider in our definition?

2. How does one measure the effect on the obtained clusters? What would constitute a "gross" effect, i.e. what kind of ΔC_1 would be considered too big for the algorithm to be stable?

We will attempt to answer both questions in the following.

Firstly, it is easy to see that the definition of "slight modification" is dependent on the SCA being examined. For example, if the SCA is based on the naming of the system's resources (as in [AL97]), then "slight modification" refers to the introduction of new elements in the system's name space, or the modification of names of existing resources. At the same time, a change in the dependencies between resources would have no effect on such an algorithm.

However, most SCAs are based on structural criteria, i.e. they view their input as a directed graph where the nodes represent the system's resources (such as files or procedures), and the edges represent dependencies between these resources (such as procedure calls or data references). Concentrating on such SCAs, we distinguish the following types of possible modifications:

1. *Edge deletion.* A dependency between two existing resources is removed.
2. *Node deletion.* A resource is removed from the system (this is usually accompanied by a number of edge deletions).
3. *Edge addition.* A dependency between two existing resources is introduced.
4. *Node addition.* A new resource is introduced to the system (this is usually accompanied by a number of edge additions).
5. *Weight modification.* This applies to SCAs that view their input as a weighted graph, e.g. the weight of an edge is equal to the number of calls between two procedures.

Moreover, it is important to decide what extent of changes we are dealing with. For the purposes of the stability study, we only consider changes that could occur during the course of a few days of development effort. Therefore, we are not concerned with stability issues between subsequent releases of large software systems (it is possible that the structure of a large software system changes drastically between releases). Section 4.3 presents a quantitative measure of the amount of modification we consider.

As for the second question raised earlier, we need a way to measure the effect a modification has on the obtained clusters, i.e. measure the size of ΔC_1. An appropriate

metric for hierarchical clustering algorithms was presented in [LG95]. However, in order to include all types of algorithms, we will adopt the MoJo metric that was presented in the previous chapter.

The distance values that will be considered acceptable for an SCA to be deemed stable will depend on the amount of input modification and the size of the software system in question. A quantitative characterization of this is given in Section 4.3.

Finally, the problem of determining the stability of an SCA can be approached from a number of different angles. We distinguish and discuss the following:

- *Analytical stability.* This refers to examining the intrinsics of each algorithm and proving that its output will not be significantly affected with any perturbation of its input. Some graph theoretic clustering methods were examined in that sense in [RY81]. However, the way an SCA works is not always known (as could be the case with commercial tools). Also, several types of approaches, such as the ones based on genetic algorithms [MMR+98], do not lend themselves well to such analysis. As a result, we did not choose such an approach.

- *Practical stability.* An SCA can be deemed stable with respect to a given software system if it has been producing stable results for this system for a long period of time. However, for the purposes of a stability study, it is impractical to try a number of SCAs with a variety of large systems over extended intervals of time.

- *Black-box stability.* This approach treats the SCA as a "black box". It provides randomly perturbated versions of an example system as input[1], and it measures the

[1]The perturbation is performed in a way that simulates real-life modifications as much as possible.

difference between the obtained clusters and the ones obtained from the original version of the system. By repeating this a number of times, one gets a good idea of an SCA's stability. This approach has the advantage that it can be applied on any SCA, and it only requires "snapshots" of large systems. For these reasons, it was the method chosen for this study.

In the next section, we present a black-box stability measure that will be used to assess the stability of various clustering algorithms.

4.3 A stability measure

Measuring the stability of an SCA entails observing its output on two different versions of the same system that are a few days of development apart. However, applying an SCA of interest on subsequent versions of several large software systems is rather impractical. For this reason, our measure (called RaSta for **Ra**ndomized **Sta**bility measure) will operate as follows:

First, it will apply the SCA in question (denoted by A) on a "snapshot" of an example system S_0 (any release of the system could serve for this purpose). This will produce a clustering C_0 of the system's resources. Then, the example system will be modified slightly in a random way. The SCA will be applied on the modified system S_1 and a clustering C_1 will be obtained. If the MoJo distance between clusterings C_0 and C_1 (we will call it $\Delta C_1 = MoJo(C_0, C_1)$) is small, then the SCA is deemed to have behaved in a stable fashion in this experiment.

The above procedure is repeated a large number of times. Each time, the original system S_0 is modified into a new system S_i, and the corresponding ΔC_i is computed.

The percentage of these experiments in which the SCA behaves in a stable fashion will be the measure of the SCA's stability with respect to the example system. We will use the notation $\Sigma\tau(A, S_0)$ to denote the stability of algorithm A with respect to software system S_0. For example, if A is 75% stable with respect to S_0 we would write $\Sigma\tau(A, S_0)$ =75%. Repeating the experiment with a number of different systems can give us a good idea of an SCA's inherent stability.

We should note that one can hardly expect an SCA to be shown to be 100% stable using the RaSta measure. The reason for this is that the random modifications might affect vital parts of the system's structure leading the algorithm astray. A stability value of 80% or more should be adequate to characterize an SCA as stable.

Certain parts of the process described above need to be defined more specifically. For instance, the magnitude of the "slight modification" has to be decided. Experience indicates that a day's development on a large software system cannot impact more than 1% of the system's resources and dependencies. Moreover, as mentioned in Section 4.2, there exist different types of possible modifications. Since development involves adding or deleting resources more often than modifying dependencies between them, we propose the following breakdown on the random modifications[2]:

- 50% are *node additions* (based on the assumption that evolving systems usually grow in size). The new nodes have to be connected to the existing ones, because otherwise they will have no effect on the algorithm's output. A random number of edges (between 1 and 0.25% of the number of resources) is added to connect each new node to existing ones.

[2]Since the algorithms we experimented with did not involve weighted edges, we have not included *weight modification* in this schema.

- 25% are *node deletions.*
- 15% are *edge additions* (experience shows that developers are more likely to introduce new dependencies rather than remove them).
- 10% are *edge deletions.*

We also need to define what we mean by small value of $\Delta C_i = MoJo(C_0, C_i)$. An upper bound for the value of the MoJo distance between two clusterings of the same set of resources is the cardinality of this set (we denote it by N). If the actual value obtained in each random experiment is less than 1% of N we will deem the SCA to have behaved in a stable fashion.

An extended version of the MoJo metric was used for this study, since the node additions and deletions result in the two clusterings being compared containing different sets of resources. This version considers only the intersection of the two sets of resources which is always at most 1% different than the set of resources of S_0.

The following describes the RaSta measure in pseudo-code. S is the software system in question, and N is the number of its resources. The number of times we repeat the random part of the experiments is denoted by the parameter TIMES.

Pseudo-code description of the RaSta stability measure

Apply the SCA on S_0 and obtain clustering C_0

counter := 0

for i : 1 .. TIMES

  Delete up to $\lceil 0.1 \frac{N}{100} \rceil$ edges from S_0

```
    Delete up to ⌈0.25 N/100⌉ nodes from S_0
    Add up to ⌈0.5 N/100⌉ nodes to S_0
    Connect each of these nodes to ⌈0.25 N/100⌉ other nodes
    Add up to ⌈0.15 N/100⌉ edges to S_0
    Apply the SCA on the modified S_0 and obtain clustering C_i
    Compute m = MoJo(C_0, C_i)
    if m ≤ ⌈N/100⌉ then
        counter := counter + 1
    end if
end for
The SCA is counter/TIMES × 100 % stable with respect to S_0
```

In the next section we present an implementation that computes the value of our stability metric, as well as some experiments we performed with it.

4.4 Implementation and Experiments

In order to test our stability measure we decided to experiment with different algorithms and software systems. We used two large software systems called TOBEY and Linux (details about these systems can be found in appendix A). Our choice of clustering algorithms was limited by what is available publically as an implementation that can run in batch mode (since we need to repeat the random experiments a large number of times). We chose two systems that are presented in appendix B, HCAS and Bunch.

For our experiments, we chose six versions of hierarchical clustering algorithms that use the Jaccard coefficient. They are shown on table 4.1. Our rationale for choosing

Algorithm name	Update rule	Cut-point height
$SL65$	Single linkage	0.65
$SL75$	Single linkage	0.75
$SL90$	Single linkage	0.9
$CL65$	Complete linkage	0.65
$CL75$	Complete linkage	0.75
$CL90$	Complete linkage	0.9

Table 4.1: Characteristics of the six algorithms chosen for our experiments.

these particular algorithms was that we would like to explore:

- The difference between complete linkage and single linkage. Complete linkage is known to produce more compact clusters. Raghavan and Yu [RY81] showed that graph theoretic clustering methods that identify more compact clusters are not as stable. We were interested to see whether that would also be true for hierarchical clustering algorithms.

- The impact of the cut-point height value. One of the hardest problems associated with hierarchical clustering algorithms is the choice of the cut-point height value. We wanted to determine the impact it has on the algorithm's stability, something that could possibly indicate a criterion for an effective choice of its value. Common values are larger than 0.5 and less than 1, so we chose the values 0.65, 0.75, and 0.9.

Also, we experimented with two of the algorithms available within Bunch, NAHC and

SAHC. We were interested to determine how stable an algorithm that includes randomization can be (Bunch starts with an initial random partition that it later attempts to improve using several optimization criteria).

In order to perform the slight modifications necessary for our experiments, we developed a program called *Factshaker* that randomly modifies files in the input format of the chosen algorithms. *Factshaker* can perform any number of modifications. For these experiments, the amount of modification discussed in section 4.3 was chosen.

The experiments were performed as follows: The source code of the input systems was used only once in order to extract the relevant information[3] in the appropriate format. The initial clustering was obtained next. Finally, *Factshaker* was employed to modify the input and obtain subsequent clusterings. The random part was repeated 1000 times for each combination of algorithm and input system.

The results concerning the two algorithms that come with Bunch were rather discouraging. When Linux was used as input, the closest that the subsequent clusterings came to the original one was a distance of 221, a value significantly larger than the stability threshold value of 10. When this was indicated to Bunch's developers, they were able to discover a bug in their implementation. However, even with the updated version of Bunch, our results indicated that both NAHC and SAHC are rather unstable algorithms.

In an effort to improve the results, we increased the population size from 1 to 12. This made the experiments run 12 times slower, but it did not improve the stability results significantly. Even when we removed the random modifications, i.e. ran Bunch with the same input 1000 times we obtained similar results. These results indicated to us that the

[3] In both cases the information extracted referred to source files and their dependencies.

	$SL65$	$SL75$	$SL90$	$CL65$	$CL75$	$CL90$
TOBEY	98.2%	99.1%	100%	74.1%	44.6%	12.5%
Linux	95.4%	97.9%	100%	64.7%	37.1%	10.4%

Table 4.2: Stability values obtained for the algorithms we examined.

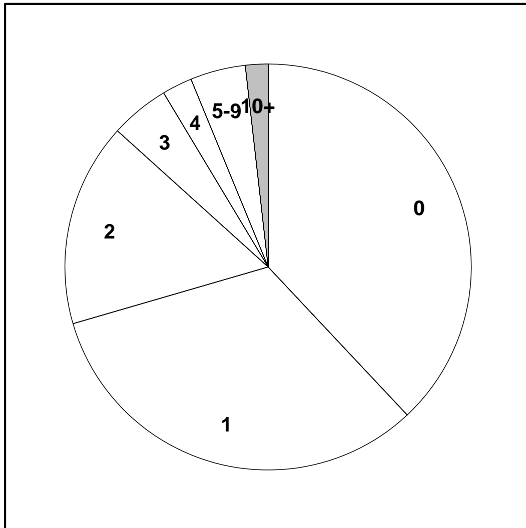
randomized part of the NAHC and SAHC algorithms causes them to be unstable and is a liability for Bunch's usability.

The results of our experiments with the six different versions of hierarchical clustering algorithms can be seen in the next three pages in the form of pie charts. The legend of each pie chart identifies the algorithm and the input system it refers to. It also presents the observed stability of the algorithm with respect to this input system. Each pie slice corresponds to a specific MoJo distance value or a range of such values. If these values indicate stable behaviour for the algorithm in question, then the pie slice is white, otherwise it is shaded (for both input systems single digit values happen to be stable, while double digit ones are unstable).

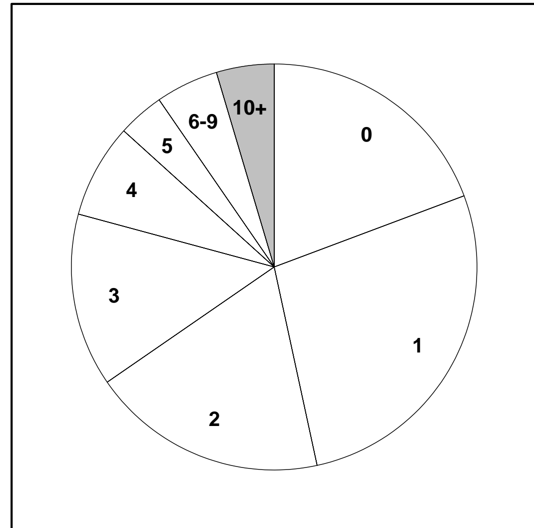
For example, figure 4.3c shows that in 74.1% of the experiments performed using the CL65 algorithm, the MoJo distance computed was less than 10 (half of these values were between 1 and 3). Also, even though 25.9% of the experiments showed CL65 to be unstable, one can see that in half of these, the value was between 10 and 14, i.e. very close to being stable.

Table 4.2 summarizes the stability values obtained. Several observations can be made:

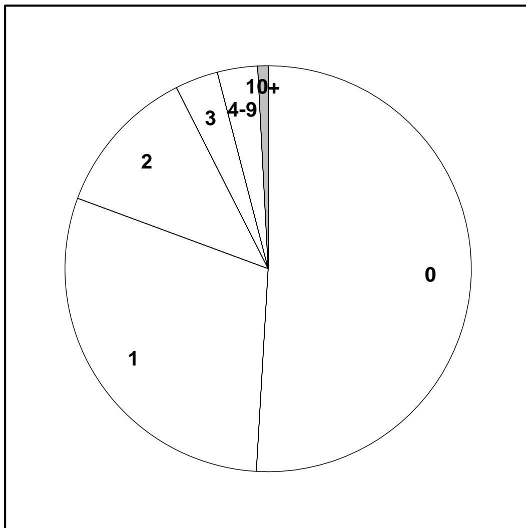
- The various algorithms seem to behave in a similar fashion with respect to the two input systems. In other words, algorithms that seem to be more stable for TOBEY, are also more stable for Linux, and vice versa. This suggests that our measure can



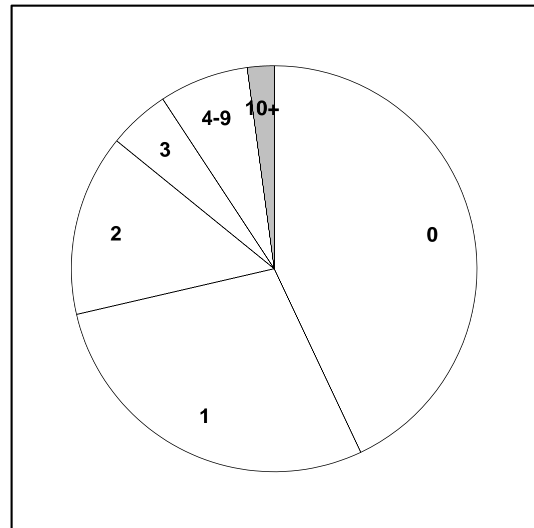
(a) $\Sigma\tau(SL65$,TOBEY)=98.2%



(b) $\Sigma\tau(SL65$,LINUX)=95.4%

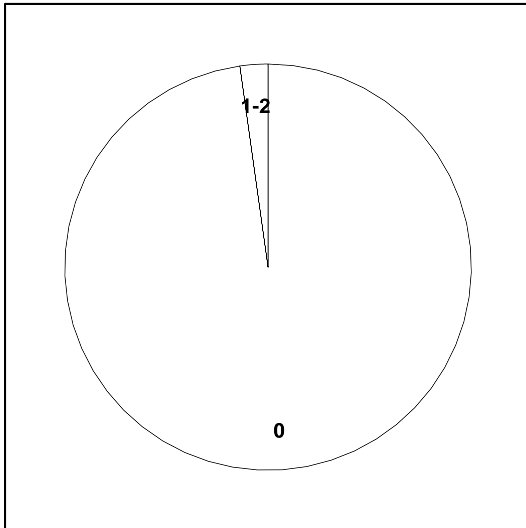


(c) $\Sigma\tau(SL75$,TOBEY)=99.1%

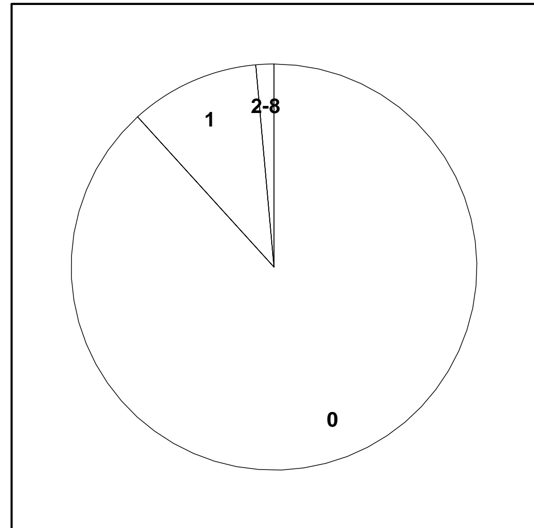


(d) $\Sigma\tau(SL75$,LINUX)=97.9%

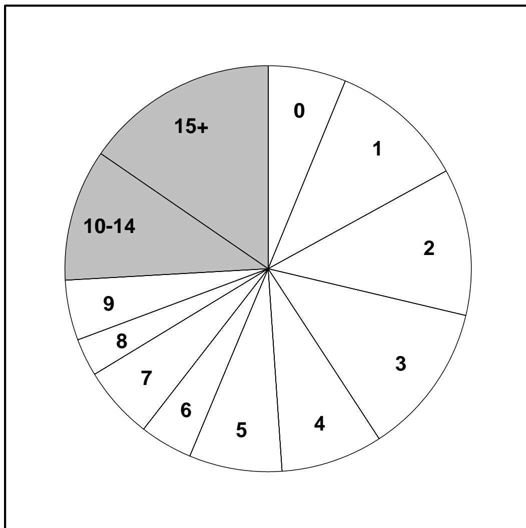
Figure 4.2: Stability values of SL65 and SL75.



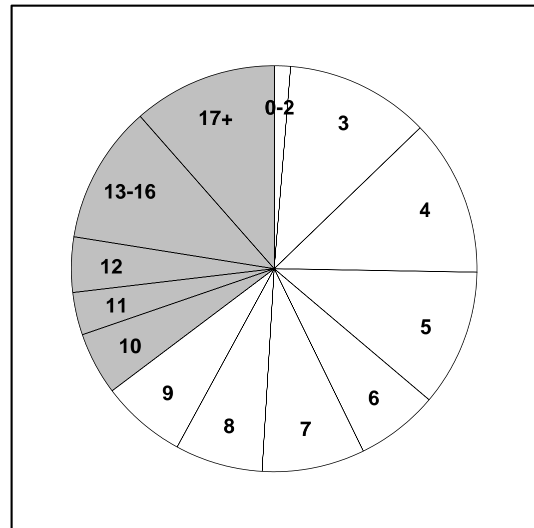
(a) $\Sigma\tau(SL90,\text{TOBEY})=100\%$



(b) $\Sigma\tau(SL90,\text{LINUX})=100\%$

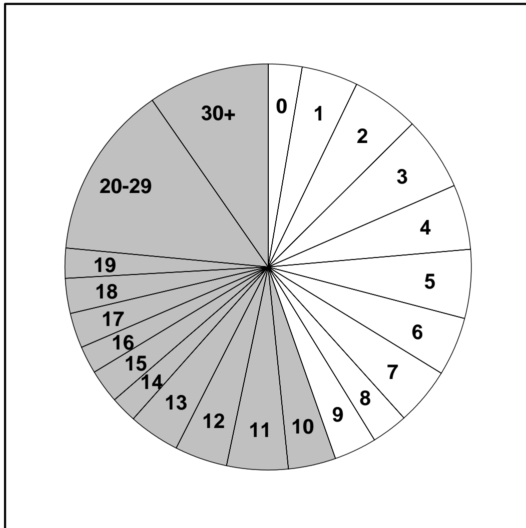


(c) $\Sigma\tau(CL65,\text{TOBEY})=74.1\%$

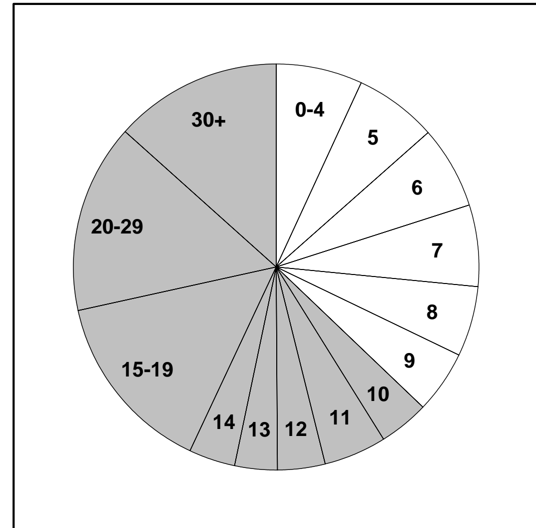


(d) $\Sigma\tau(CL65,\text{LINUX})=64.7\%$

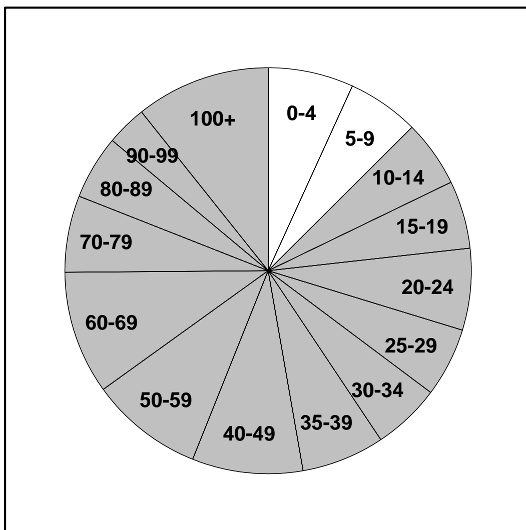
Figure 4.3: Stability values of SL90 and CL65.



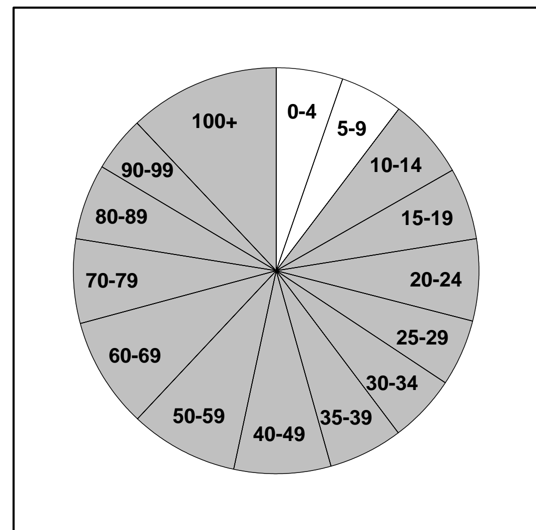
(a) $\Sigma\tau(CL75$,TOBEY)=44.6%



(b) $\Sigma\tau(CL75$,LINUX)=37.1%



(c) $\Sigma\tau(CL90$,TOBEY)=12.5%



(d) $\Sigma\tau(CL90$,LINUX)=10.4%

Figure 4.4: Stability values of CL75 and CL90.

give us a good idea of an algorithm's stability.

- The stability values obtained for Linux are consistently slightly lower than the ones obtained for TOBEY. This is true even for algorithm $SL90$ where both values are 100% (contrast figures 4.3a and 4.3b). This discrepancy seems rather strange at first since the graphs of the two systems have roughly the same number of nodes and edges (the TOBEY graph is composed of 939 nodes and approximately 9,000 edges, while the Linux one has 955 nodes and around 10,000 edges). However, closer examination reveals that TOBEY contains more nodes with small degree, while many Linux nodes have a rather large degree (Figure 4.5 presents a graph that demonstrates this fact). This would suggest that the deletion of a node from the Linux graph would be accompanied by the deletion of a larger number of edges on average, which could explain larger discrepancies in the SCA's output making it appear less stable.

- For both example systems, it appears that a hierarchical clustering algorithm using the single linkage update rule is more stable than one using the complete linkage update rule. This result agrees with the theoretical predictions by Raghavan and Yu.

  The wide difference in the values between corresponding algorithms (for example, $SL90$ and $CL90$) can be explained by the algorithm's nature. An algorithm using single linkage tends to form clusters much earlier than one using complete linkage. As a result, when the cut-point height was set to 0.9, the single linkage algorithm would produce a small number of clusters (typically 5-7) while the complete linkage one would find many more (consistently over 100). Fewer, better-formed clusters

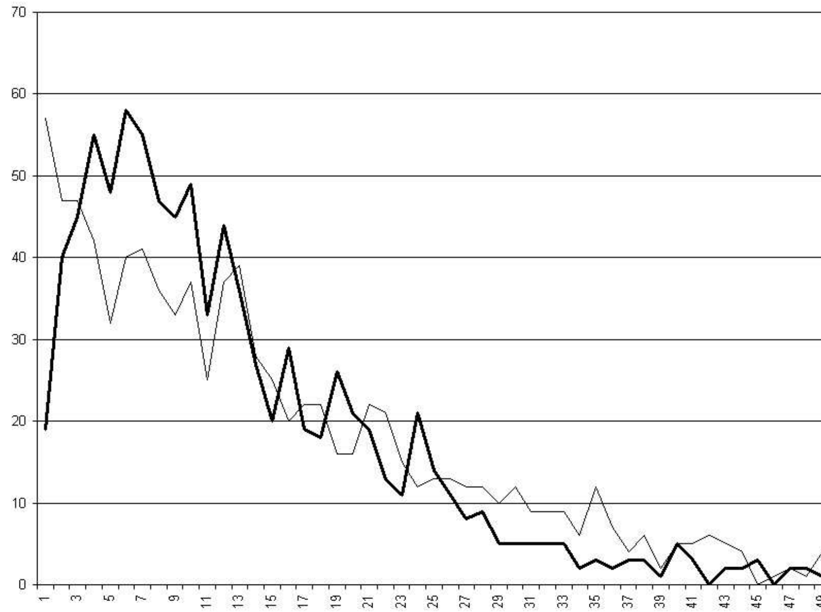


Figure 4.5: Node degree distribution for TOBEY (thick line) and Linux (thin line). A point's x-value is the node degree (total number of incoming and outgoing edges) and the y-value is the number of nodes in the system's graph that have this degree.

can easily explain the difference in the stability values obtained.

- It appears that as the cut-point height of a hierarchical clustering algorithm using the complete linkage update rule increases, the stability of the algorithm decreases (at least up to the value of 0.9). That was a rather unexpected result since in theory an algorithm with a cut-point height of 1 should be 100% stable (a cut-point height of 1 means that all objects are grouped in one cluster). Further research is required to determine the stability behaviour of such algorithms with cut-point height between 0.9 and 1.

- Our confidence in our stability measure has increased after these experiments, since the experimental results seem to agree with the theoretical ones, and they do not seem to be affected by the input system. At the same time, these results indicate that stability could be used as a means of comparison of different SCAs, as suggested by Jain and Dubes [JD88]. Although it is definitely not the only deciding factor[4], it can provide valuable insight to the usefulness of an SCA.

4.5 Conclusions

This chapter introduced the issue of stability for SCAs. We presented reasons why we believe this is an important issue, and why we consider stability to be a desirable feature for an SCA.

We also presented a measure that evaluates an SCA in terms of its stability. We performed several experiments with this measure. The results indicate that our metric can be a good predictor of an algorithm's stability.

In the next chapter, we will present an alternative way of addressing the problem of keeping the decomposition of an evolving software system up-to-date.

[4]Other factors might include the type of clustering an algorithm produces, or features such as cluster naming.

Chapter 5

Clustering evolution

Industrial software systems are constantly under development. Their structure may fluctuate from day to day due to the creation of new resources (such as new procedures or source files) or the restructuring of existing ones. This creates an interesting problem regarding the current decomposition of a given software system: can one keep it up-to-date in an automatic way?

In this chapter, we concern ourselves with the effect of software evolution on the structure of a software system. We identify two interesting subproblems, namely that of assigning a newly introduced resource to a subsystem (incremental clustering), and that of accommodating structural changes (corrective clustering). We propose an algorithm to handle these problems and validate it by presenting three case studies.

The next section presents the problem in more detail and argues about the importance of finding a satisfactory solution.

5.1 The Orphan Adoption problem

The software clustering literature contains many techniques that attempt to decompose software systems in an automatic way. Reverse engineering or redocumentation tools, such as the Software Bookshelf [FHK+97], can then be employed in order to present this structure to the developers in an effective, visual fashion. However, if such a tool is to be used in conjunction with the software development process, it needs to reflect the structure of the software system as it evolves. In other words, changes to the structure of the system have to be accommodated on a continuing basis.

We identify the following two problems as being important for any reverse engineering project dealing with a software system under development:

1. *Orphan adoption.* We define an *orphan* as a newly introduced resource (e.g. procedure, variable or source file) to the system. An orphan adoption process attempts to place the orphan in an appropriate existing subsystem. This process is an example of *incremental clustering.*

2. *Kidnappee adoption.* Development effort on a software system sometimes introduces structural changes that might demand that a resource be moved from its original subsystem to a new one. This is effectively equivalent to "kidnapping" a resource from its parent subsystem — thereby making it an orphan — and then readopting it to the appropriate subsystem. This process is an example of *corrective clustering.*

In the following we will refer to both versions of this problem as "the Orphan Adoption problem".

An effective solution to this problem is important to reverse engineering projects for the following reasons:

- The visualizations of the software system's structure that are presented to the developers have to be up-to-date, otherwise they are misleading and confusing.
- The layout of a visualization may convey important semantic information to the developers (e.g., the way procedures are laid out can imply their order of execution). Therefore, it is important to solve the Orphan Adoption problem in a way that ensures minimal effect on the software system's structure and its corresponding visualizations.
- A corrective clustering algorithm can be used not only in order to "readopt kidnapped resources", but also to indicate candidates for restructuring. This in turn can lead to bug detection or to structural improvements that will increase the maintainability of the system.

As a result, there exists a need to devise criteria for adopting or readopting orphans. Many types of criteria can be considered. We distinguish and discuss the following categories:

1. *Naming criteria.* These criteria rely on the fact that a naming convention for resource names might be sufficient to identify the subsystem to which a resource belongs. It is very common for large software products to follow such a naming convention, (e.g. the first two letters of a file name might identify the subsystem to which this file should belong).

Such criteria are, of course, specific to each software project, but they are typically rather simple and easy to automate. The disadvantage of using naming criteria is that they rely on the developers to follow the naming convention. Experience shows that this is not always the case.

2. *Structural criteria.* In any software system, one can define dependencies between the system's resources (such as procedure calls or data references). In this case, structural criteria can be used as a basis for determining an appropriate parent subsystem for an orphan. This is done by examining the way the orphan interacts with the contents of the candidate subsystems.

   Typical examples of structural criteria include adoption of the orphan by the subsystem it displays the largest connectivity with [Nei96], or by the subsystem whose interface size (number of externally visible resources) will decrease the most upon the incorporation of the orphan.

3. *Stylistic criteria.* These criteria attempt to take into account the style with which developers create or researchers decompose a system. For example, two kinds of subsystems are commonly identified in most systems [CTH95]: ones that implement a part of the functionality of the system (these subsystems contain resources that interact mostly with each other, and therefore exhibit high cohesion and low coupling), and ones that provide utilities for the rest of the system (the contents of such subsystems might not interact with each other at all, while they might be accessed by the majority of the remaining subsystems, and therefore exhibit low cohesion and high coupling). A stylistic criterion attempts to classify an orphan as a utility or not, and adopt it accordingly.

4. *Semantic criteria.* Criteria of this type attempt to adopt orphan resources by analyzing their source code and deciding to which part of the functionality of the system they refer. For example, a semantic criterion might determine that a newly introduced file implements a user interface function. It will then place it in the corresponding subsystem. The automatic use of semantic criteria needs to employ program understanding techniques. For this reason, such criteria are difficult to implement.

In the next section we present an algorithm that addresses the Orphan Adoption problem. We also discuss the criteria we used, as well as the order in which they were applied, since it is possible that different criteria might yield different results for the same orphan.

5.2 The Orphan Adoption Algorithm

Utilizing a number of criteria such as the ones presented in the previous section, we developed an algorithm that addresses both sub-problems of the Orphan Adoption problem. The main flow of the algorithm (called the Orphan Adoption algorithm) is depicted in figure 5.1.

5.2.1 Overview

The Orphan Adoption algorithm accepts as input the name of a resource, which might be either a newly introduced resource or a kidnapped one. It emits as output the name of the subsystem that has been chosen as an appropriate parent for the orphan. Our algorithm also has access to the following information:

- The current decomposition of the software system in question.
- The stylistic designation of each subsystem, i.e. whether it is a utility subsystem or not.
- The naming convention used in the particular software system.
- The dependency graph of the system.

A diamond-shaped node in the Orphan Adoption algorithm flowchart contains a condition that has to be evaluated in order to determine the direction in which the flow of the algorithm will proceed next. A number of predicates are contained in some of the conditions in the flowchart of the Orphan Adoption algorithm. Their meaning is the following:

- The predicate KIDNAPPED is true if the orphan was already part of a subsystem.
- The predicate UTILITY is true if the parent of a "kidnapped" orphan was stylistically designated as a utility subsystem.
- The predicates "#=1" and "#=2" are true if the cardinality of the parent subsystem was respectively one or two.

Two or more predicates in the same condition are implicitly joined by an OR operator.

As can be seen in figure 5.1, our algorithm initially determines whether any kidnapped orphans belonged to subsystems of low cardinality or to utility subsystems. If that is the case, they are re-assigned to the same parent. The reasoning behind this is the following: the contents of small subsystems will almost always be adopted in other subsystems using

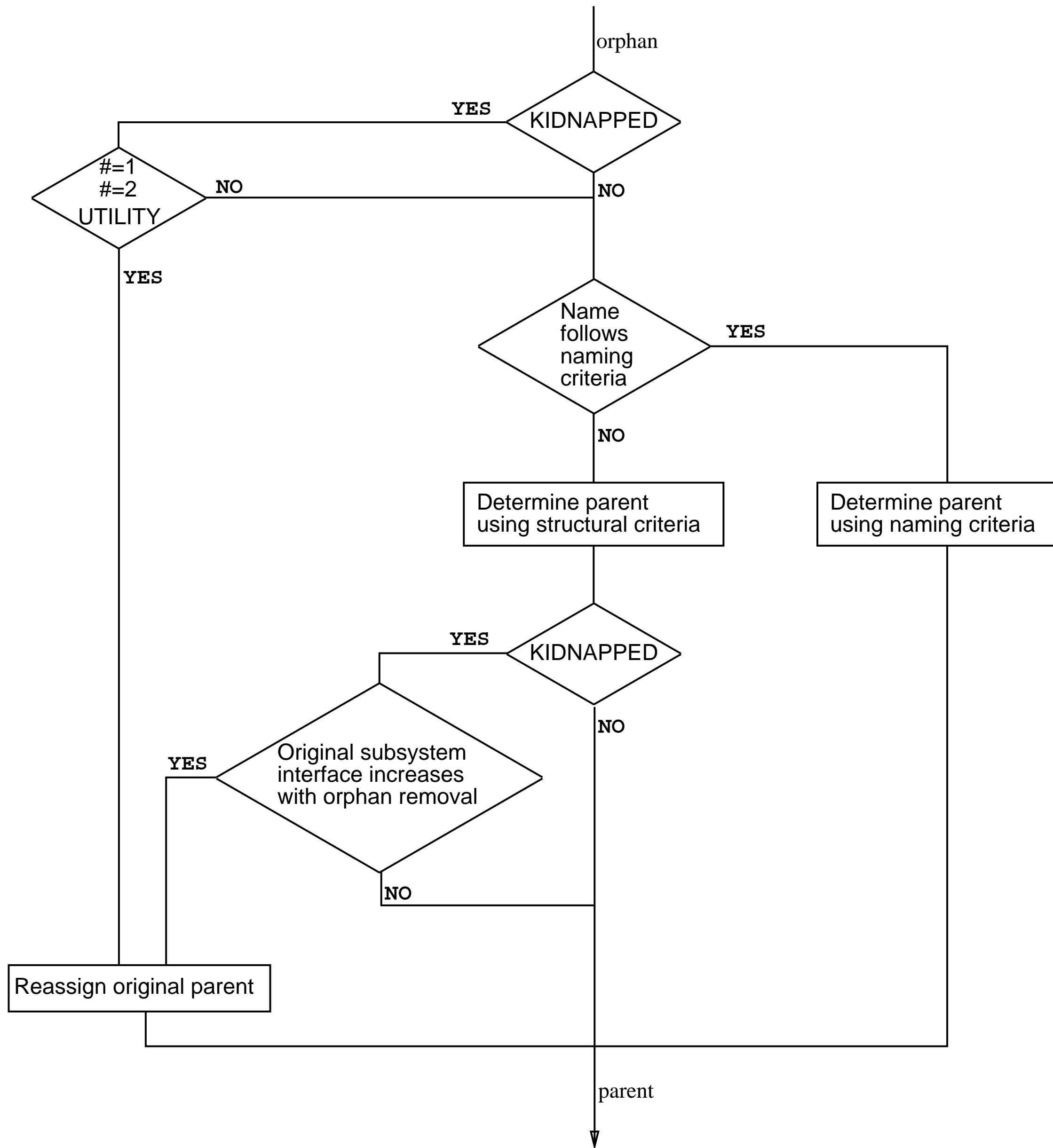


Figure 5.1: The flowchart of the Orphan Adoption algorithm.

structural criteria, since they cannot have any strong ties to their subsystem. However, if these subsystems exist in the decomposition, it probably means that the developers feel that they should be there because of their semantic importance. A similar argument holds for contents of utility subsystems.

Next, our algorithm determines whether the name of the orphan conforms to any known naming conventions. If so, the orphan is adopted by the appropriate subsystem. However, many software systems do not have any naming conventions, or the developers simply do not follow them. In this case, structural criteria (described in more detail in the next section) are employed to determine the appropriate parent for the orphan.

Finally, if the orphan has been kidnapped, there is an extra condition that has to be met before the orphan is assigned to a new parent. This is explained in more detail in section 5.2.4.

5.2.2 Structural Criteria

The use of the structural criteria requires further elaboration. In order to present them in a formal fashion, let us define the following:

- Let $S = \{s_i | i : 1..n_s\}$ be the set of all subsystems and $R = \{r_i | i : 1..n_r\}$ be the set of all resources. Note that since subsystems are also considered to be resources, we have $S \subset R$. We define the relation $p \subseteq R \times S$, such that the pair (r_i, s_j) belongs in p, iff the subsystem s_j is the parent of resource r_i in the current subsystem hierarchy. We write $p(r)$ to denote the parent of resource r.

- Let $d = \{d_j | j : 1..n_d\}$ be the set of all defined dependencies between resources (such as procedure call or source inclusion). Each dependency d_j is a set of ordered pairs.

The first element of each ordered pair is related to the second one with dependency d_j. For example, if d_3 denotes source inclusion and $d_3 = \{(r_1, r_2), (r_3, r_4)\}$, then resource r_1 includes resource r_2 and resource r_3 includes resource r_4.

We define $D = \cup_{j=1}^{n_d} d_j$. We write $r_1 D r_2$ if resource r_1 depends on resource r_2.

- Let $W(s, r)$ be the weight of the induced dependency between a subsystem s and a resource $r \notin s$. More formally, $W(s, r) = |\{u | p(u) = s \wedge uDr\}|$.

The structural criteria in our algorithm are used to assign a parent to an orphan o as follows:

$$p(o) = s_k, \text{ if } W(s_k, o) \geq W(s_i, o), \forall i \in 1..n_s$$

In other words, the algorithm places the orphan in the subsystem that seems to depend most on it (tie-breakers are discussed in the next section). This criterion was deemed superior to alternative ones that examine also subsystems that the orphan depends on. This was based on the assumption that a newly introduced resource of a non-utility subsystem will primarily provide facilities for other members of its subsystem, while it can just as well depend on resources belonging to different subsystems. Common programming practice and our experience with different systems suggest that this is usually true.

5.2.3 Tie-breakers

In order to deal with cases where two or more subsystems exhibit the same maximum weight of dependency to the orphan, our algorithm employs two tie-breakers. In order of precedence, they are:

1. Let us define the interface of a subsystem s as the set $I(s)$, where $I(s) = \{r | p(r) = s \land \exists r' : p(r') \neq s \land r'Dr\}$. The addition of the orphan to one of the candidate subsystems might decrease the cardinality of its interface (note that the orphan will always be part of the interface in this case, but its addition might "hide" some members of the previous interface). The subsystem whose interface decreases the most is preferred over the other candidates.

2. The set of supplier subsystems for a subsystem s is defined as $Sup(s) = \{s' | s' \neq s \land \exists r : p(r) = s \land \exists r' : p(r') = s' \land rDr'\}$. Our algorithm will choose to adopt the orphan in such a subsystem, so that its supplier set is increased as little as possible.

If none of the tie-breakers manage to resolve the tie, then the orphan is adopted arbitrarily by one of the candidate subsystems.

5.2.4 Interface Minimization

The aforementioned assumption, that an orphan o will primarily provide facilities for other members of its subsystem, is generally true only if o is a newly introduced resource to the system. If o is a kidnapped resource, this assumption does not necessarily hold, since it is possible that the orphan is a point of entry to its subsystem, and that only external resources depend on it. In this case, its removal from the subsystem would be ill-advised.

For this reason, if our algorithm determines that a kidnapped resource (initially contained in subsystem s_{old}) should be adopted by a different parent subsystem s_{new}, the move will happen only if the following condition holds: the cardinality of the interface of subsystem s_{old} must not increase as a result of the orphan's removal.

The reasoning behind this condition is the following: orphan o was part of the interface of subsystem s_{old} (otherwise it could not have been re-adopted by a different subsystem). Even though o is removed from s_{old}, its interface increases, which indicates that at least two other resources that were not previously part of the interface of s_{old} are now accessed directly from other subsystems. This strongly indicates that o was a point of entry for s_{old}, and therefore should not be removed.

5.2.5 Observations

Several interesting observations can be made about the nature of our algorithm:

- Naming criteria take precedence over structural ones in our algorithm. The underlying assumption is that if developers bother with the naming convention, then they do so in a correct manner.
- Stylistic criteria are only used for "kidnapped" orphans (if the parent was a utility subsystem or it had a cardinality of 1 or 2, then the orphan is reassigned to the original parent). A possible extension that will be better explained in chapter 6 would be to consider any orphan with an in-degree larger than 20 as a candidate for adoption by utility subsystems only.
- Our algorithm does not utilize any semantic criteria. One usually needs specific domain knowledge in order to employ such criteria, something that would reduce our algorithm's applicability.

Finally, it should be apparent that the Orphan Adoption algorithm does not always assign parent subsystems with the same level of confidence. An orphan interacting with

only one subsystem is very confidently adopted, while as we saw in section 5.2.3, the selection of parent subsystem might be arbitrary in extreme cases.

The level of confidence with which our algorithm adopts an orphan can be important information to someone that wants to ensure that the decomposition of a given software system is correct. For this reason, we developed a measure for the confidence level of the adoption of an orphan.

A naive solution would be the ratio of the orphan's interaction with the chosen subsystem to the orphan's total interaction. If we denote the chosen subsystem by s_k and the confidence level measure of the adoption of orphan o by s_k with $CL(s_k, o)$ we would have:

$$CL(s_k, o)_{naive} = \frac{W(s_k, o)}{\sum_{i=1}^{n_s} W(s_i, o)}$$

This would work rather well in the following example: Orphan o interacts with only two subsystems s_1 and s_2, and we have $W(s_1, o) = W(s_2, o) = 5$. If our algorithm chooses either subsystem as the parent, the confidence level can be computed at 0.5, which seems appropriate.

However, let us consider another example: Orphan o interacts with six other subsystems, s_1 to s_6. We have $W(s_1, o) = 5$, and $W(s_j, o) = 1, j : 2..6$. In this case, s_1 seems to be the appropriate parent for o. However, the confidence level would still be 0.5 which seems rather unfair. For this reason, we have added an extra term to our measure to increase its value when the mean dependency weight of the non-chosen subsystems is considerably smaller than the dependency weight of the chosen subsystem. This term is multiplied by $1 - CL(s_k, o)_{naive}$ to ensure that the final result will be between 0 and 1. The formula for our metric is now:

$$CL(s_k, o) = CL(s_k, o)_{naive} + (1 - CL(s_k, o)_{naive}) \frac{W(s_k, o) - \frac{\left(\sum_{i=1}^{n_s} W(s_i, o)\right) - W(s_k, o)}{n_s - 1}}{W(s_k, o)}$$

This is actually equivalent to the following, which represents the final formula of our confidence level measure (presented as a percentage):

$$CL(s_k, o) = \left(1 - \frac{\left(\left(\sum_{i=1}^{n_s} W(s_i, o)\right) - W(s_k, o)\right)^2}{(n_s - 1) W(s_k, o) \sum_{i=1}^{n_s} W(s_i, o)}\right) \times 100\%$$

As can be easily verified, the confidence level for our first example is still 50%, while the one for the second example is now up to a more appropriate 90%. It is interesting to note that the confidence level reaches 100% only when there exists exactly one subsystem that depends on the orphan.

In the next section, we present three case studies that demonstrate the effectiveness of the Orphan Adoption algorithm.

5.3 The Case Studies

In order to evaluate the effectiveness of our algorithm, we conducted three case studies with systems of various sizes. In each of these studies, the resources that are being clustered into subsystems are source files.

As shown in the results below, the effectiveness of our algorithm was established, not only in dealing with the Orphan Adoption problem, but also in identifying parts of the system that need restructuring.

5.3.1 Case study 1 : MiniTunis

MiniTunis is a toy operating system that has been used to teach the Operating Systems course at the University of Toronto. It consists of approximately 7,000 lines of code written in Object Oriented Turing. The structure of the system was well-known to us, and can be seen in figure 5.2 in a diagram produced by the Landscape tool [Pen92].

-- The Minitunis Operating System --

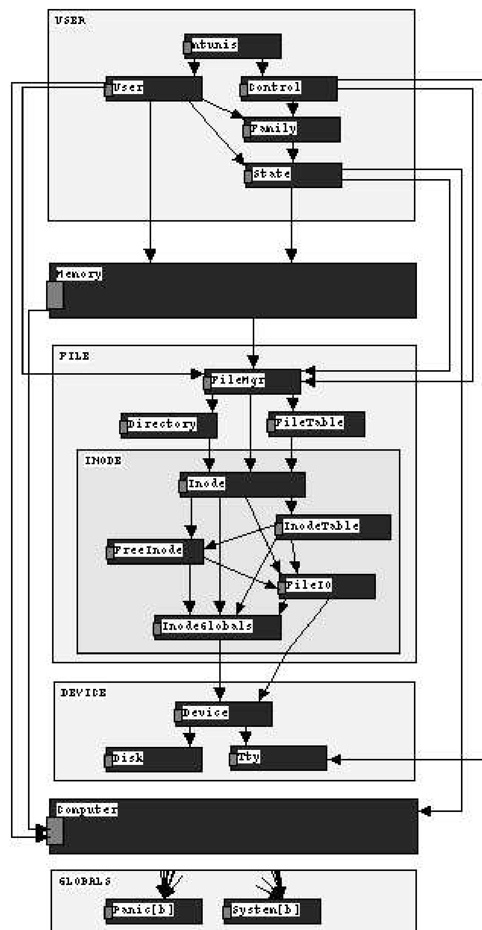


Figure 5.2: The system decomposition of the MiniTunis operating system.

Since the system is not evolving, there is no incremental clustering to be done. How-

ever, we used this case study in order to verify that the corrective clustering part of our algorithm will do a reasonable job with a system whose structure is well-known. The results were very encouraging, since each module (represented by dark boxes in the diagram) was readopted back into its original subsystem.

5.3.2 Case study 2 : Neuma

In our next case study, we considered evolving systems rather than established ones. Neuma, a company specializing in configuration management, kindly supplied us with data from 10 different versions of one of their products (approx. 80,000 lines of code each). This data contained information about source filenames, source inclusion relations between files and system decomposition for each version.

In the experiment we conducted, we took the system decomposition for version $n-1$ and the source inclusion data for version n, and attempted to come up with a system decomposition for version n by adopting all the orphans that version n introduced. By comparing the output of our algorithm to the actual decomposition of version n provided by the developers, we were able to evaluate the effectiveness of our algorithm.

Out of 17 orphans introduced in the system at any point during its evolution, 14 of them were adopted correctly (a percentage of 82%). Also, the corrective clustering part of our algorithm, when applied to all 10 versions, readopted again all resources to the same subsystems from which they had been "kidnapped".

5.3.3 Case study 3 : TOBEY

Our last case study was TOBEY. As described in more detail in appendix A, TOBEY was clustered manually by a seasoned developer. For the first part of this case study, we were interested to see whether the corrective clustering part of the Orphan Adoption algorithm would suggest any modifications to this decomposition.

Accordingly, we kidnapped each resource from the decomposition and attempted to re-adopt it. Out of 939 source files, our algorithm suggested that 46 be placed in a different subsystem than the one suggested by the system's developer. When the developers were presented with these results, they felt that 33 out of the 46 suggestions made by the Orphan Adoption algorithm were valid. The rest of them, while not necessarily wrong, were not optimal. In some cases, the developers felt that the orphan should stay in the original subsystem, but some restructuring should take place in order to justify this.

At a later stage, we acquired data from a more recent release of TOBEY and tried to adopt the orphans found in it. Out of 89 newly introduced resources, 79 were adopted correctly according to the developers (a percentage of 88.8%).

5.3.4 Observations

Table 5.1 summarizes the results of our case studies. From the data in this table, we can make the following observations about the performance of our algorithm in dealing with the three example systems:

- Our algorithm provides a satisfactory solution to the problem of Orphan Adoption. It helps to reduce significantly the manual effort required to produce a system decomposition for a new version of a system.

system name	lines of code	dependency used	number of resources	incremental clustering	corrective clustering
MiniTunis	7,000	import	20	—	100%
Neuma	80,000	include	40-57	82%	100%
TOBEY	250,000	proc call/data ref	939	88.8%	95.1%

Table 5.1: Summary of results.

- The corrective clustering part of the algorithm makes valid suggestions for resource migration, and it is able to identify parts of the system that might be in need of restructuring.

- Our algorithm seems to be effective regardless of the type of dependency used, since it handled three different cases successfully (import relations for MiniTunis, source inclusion for Neuma, procedure calls and data references for TOBEY).

Another interesting observation is that all the criteria utilized by our algorithm seem to contribute to its effectiveness. To demonstrate this, we identified the criterion that was responsible for each correct result of our algorithm. Table 5.2 presents a breakdown of the correct results of our algorithm for each of the three case studies. Even when naming conventions were not available, as in the case of the first two systems, the rest of the criteria share the task of correctly (re)adopting orphans.

	MiniTunis	Neuma	TOBEY
Naming Criteria	0%	0%	63.9%
Structural Criteria	65%	45.7%	15.2%
Stylistic Criteria	20%	0%	18.7%
Interface Minimization	15%	54.3%	2.2%

Table 5.2: Breakdown of correct results by criterion category.

In this chapter, we presented the problem of Orphan Adoption that occurs during the evolution of a software system. It is a problem that reverse engineering tools need to solve effectively in order to support the software development process. We proposed an algorithm to solve this problem, and described case studies that suggest that the algorithm can deal with it effectively.

In the next chapter, we present an approach that uses the Orphan Adoption algorithm to automatically cluster a software system.

Chapter 6

Pattern-driven clustering

One definition of the notion of comprehension is that of "establishing a link between what you see and something you've seen before". In that light, the process of understanding can be paralleled to that of pattern recognition.

It has been successfully argued [GHJV95] that the use of patterns can be of considerable help to software developers. However, the patterns one employs when developing software differ from the patterns one might utilize when attempting to understand an existing software system. In the former case, one deals with the complexity of vagueness; there is usually a large number of possible solutions to a complex software engineering problem. In the latter case, one deals with the complexity of crispness; the amount of information available is overwhelming.

In this chapter, we use patterns to address the difficult task of comprehending the structure of existing large software systems.

6.1 Introduction

Software clustering techniques presented in the literature employ certain criteria in order to decompose a software system. Such criteria include low coupling and high cohesion, interface minimization, shared neighbours etc. However, it appears that in an effort to maximize the performance or accuracy of their algorithms, many researchers have lost sight of the fact that their primary concern is to aid comprehension. Our first priority should not be to ensure that the clusters produced by our approach satisfy certain abstract criteria, but that they are effective in helping someone understand the software system at hand.

For this reason, we believe that an effective software clustering algorithm should produce output that can be easily understood. This can be achieved by proposing clusters that follow familiar patterns, by naming these clusters appropriately, and by limiting the cardinality of each cluster to manageable levels. Coupled with effective visualization techniques, such an approach can produce clusterings that are helpful to the comprehension process of a software system.

In this chapter, we present a software clustering approach that subscribes to this philosophy. It discovers clusters that follow patterns that are commonly observed in decompositions of large software systems that were prepared manually by their system architects. We also present an algorithm that follows such a pattern-driven approach, called ACDC (for an **A**lgorithm for **C**omprehension-**D**riven **C**lustering). ACDC assigns automatically meaningful names to the clusters and tends to avoid creating clusters of very large size.

The structure of this chapter is as follows: Section 6.2 presents our approach to clus-

tering for program comprehension. Common subsystem patterns are listed in section 6.3. Section 6.4 presents ACDC and its features. Section 6.5 describes several experiments we conducted with large industrial systems to demonstrate the usefulness of our algorithm.

6.2 Clustering for comprehension

Most of the algorithms presented in the software clustering literature identify clusters by utilizing certain intuitive criteria, such as the maximization of cohesion, the minimization of coupling, or some combination of the two. Such criteria can definitely produce meaningful clusterings, a fact that is showcased by the success such approaches have had with a variety of software systems [Sch91, CS90, MOTU93, MMCG99].

However, we believe that an excessive amount of effort has been put into fine tuning these techniques. In an effort to improve the accuracy or performance of their algorithms, researchers have lost sight of their primary goal, comprehension. Rather than trying to discover a clustering that exhibits slightly larger cohesion values, one should strive to cluster the software system in a way that will aid the process of understanding it.

It is a well-known fact that any clustering problem does not have a single answer [Eve93]. Depending on the point of view of the clustering procedure, different clusterings can serve as the answer. Hence, it is more important to attempt to produce a clustering that will be helpful to the developers trying to understand the software system, rather than to try to find the clustering that maximizes the value of some metric.

We believe that a software clustering algorithm should have certain features that will ensure that its output will be helpful to someone attempting to understand the software system at hand. These features include:

- *Effective cluster naming.* Providing meaningful names for the obtained clusters is an issue that has not attracted much attention (Schwanke acknowledged the importance of this problem and presented a semi-automatic solution in [SP89]). We believe that it is important that the high level view of a software system contains subsystems with familiar names, rather than names such as Subsystem01, SS02 etc. Effective naming allows people associated with the software system to understand the obtained decomposition faster. This means that they can refine it more easily, and start benefiting from it more readily.

- *Bounded cluster cardinality.* A partition of a system's resources where one cluster contains the overwhelming majority of the resources while the remaining clusters contain one or two resources each, can hardly be useful, even though it might exhibit low coupling. On the other hand, clusters containing a limited number of objects (up to 20 or so) are more manageable and easier to understand. An algorithm targeted towards program comprehension should attempt to create decompositions that contain clusters with a bound on their size. This should not be done however at the expense of the system's structure. The algorithm should simply decompose any large clusters further, producing nested clusters of more than one level if necessary.

- *Pattern-driven approach.* Humans can understand something more easily if it is presented in familiar terms or patterns. Our experience indicates that certain patterns emerge time and again when humans create manual decompositions of software systems they are knowledgeable of. This suggests that if a decomposition contains these patterns, it would be easier to understand.

  For this reason, we believe that an algorithm that identifies these patterns, and

creates a system decomposition based on them can be successful in helping the comprehension process of a software system.

The next section presents a list of subsystem patterns that are commonly observed in manual decompositions of large industrial systems.

6.3 Subsystem patterns

From city planning to object-oriented programming [GHJV95], it is well-known that patterns are followed in many a creative process. In a software reengineering context, a pattern might refer to a process that alleviates certain legacy software problems [DRN99]. When it comes to software clustering, patterns refer to familiar subsystem structures that frequently appear in manual decompositions of large industrial software systems.

The remainder of this section presents several subsystem patterns. It is important to note that these patterns are found in large systems. They might not necessarily apply to smaller systems (in some cases a minimum of about 100 source files is necessary for the pattern to emerge).

The presentation of each pattern contains four parts:

1. *Description.* This part explains how one can recognize the pattern.

2. *Diagram.* This is a visual depiction of the pattern. The following colouring scheme applies to all diagrams:

    - Green boxes represent variables.

    - Dark blue boxes represent procedures.

- Blue boxes represent source files.
- Arrows represent dependencies between resources.
- Nested boxes denote containment, e.g. a procedure that is declared in a source file, or a source file residing in a directory.
- The dashed red boxes enclose the resources that are grouped together by the pattern.

3. *Rationale.* This part presents the reasons why it is likely that the pattern will appear in large systems, as well as the reasons why we believe the pattern is effective for comprehension purposes.

4. *Instance naming.* This last part explains how instances of the pattern can be named appropriately. Subsystem names always contain the suffix ".ss".

It should be noted that no implementation details are included in the following. An algorithm that follows this pattern-driven approach may choose to implement these patterns as seen fit. Section 6.4.1 presents implementation details for our own pattern-driven clustering algorithm, ACDC.

Source file pattern

Description

Depending on the programming language used, a source file may contain the definition of one or more procedures/functions, as well as the declaration of several variables. This pattern would group all the procedures and variables contained in the same source file into one cluster.

Diagram

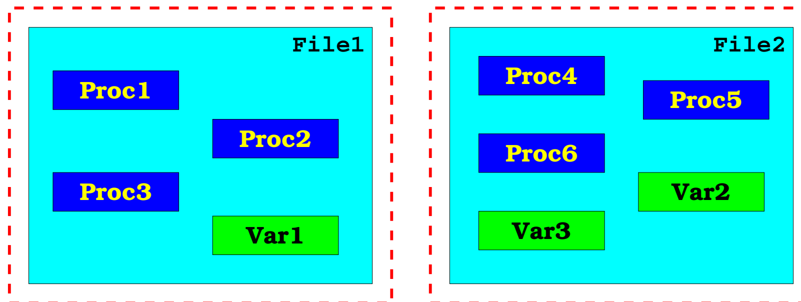


Figure 6.1: The source file pattern.

Rationale

The underlying assumption behind this pattern is that developers would not place the definition of two procedures in the same file, unless they were related. However, even if that is not true, the comprehension value of this pattern should be clear, since it reflects the structure of the source code.

Instance naming

The name of the source file with the suffix ".ss" appended to it will be the name of any clusters formed using this pattern. This should provide an intuitive and familiar name to the cluster.

Directory structure pattern

Description

The source code of many large software systems is divided into directories. This pattern groups the files contained in the same directory into one cluster.

Diagram

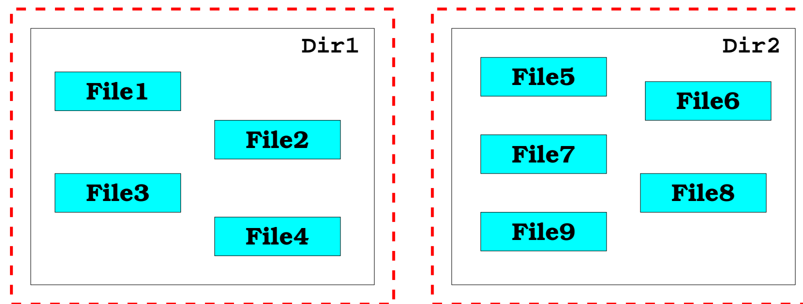


Figure 6.2: The directory structure pattern.

Rationale

It is common practice amongst developers to put related files in the same directory, which is the main reason for the introduction of this pattern. However, one has to be careful with the application of this pattern as this is not always the case. For example, many systems contain directories that are collections of header files. Such a cluster might not be useful from a comprehension point of view.

Instance naming

Again, the name of the directory followed by the suffix ".ss" should be an appropriate name for the newly-formed cluster.

Body-header pattern

Description

Many programming languages are designed in a way that results in a procedure being split between two different files, e.g. in C a header file (a .h file) and a body file (a .c file). This pattern lumps such files together into a cluster (usually containing only two

files).

Diagram

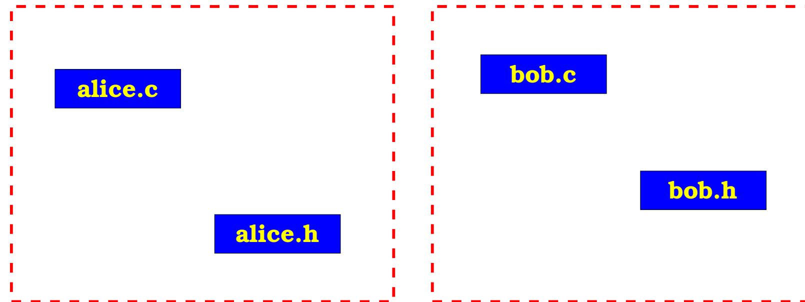


Figure 6.3: The body-header pattern.

Rationale

The clusters created by this pattern are rather special since low cardinality is acceptable, if not desirable. Their introduction can reduce the complexity of a system's structure significantly while the amount of information conveyed is intact.

Instance naming

The two constituents of the clusters formed by this pattern usually have the same name with a different extension, e.g. `parser.c` and `parser.h`. The common part of their name followed by the suffix ".ss" should be a good name for the new cluster.

Leaf collection pattern

Description

This pattern groups together resources that are leaves in the dependency graph of the software system, i.e. resources that do not depend on any other resource.

Diagram

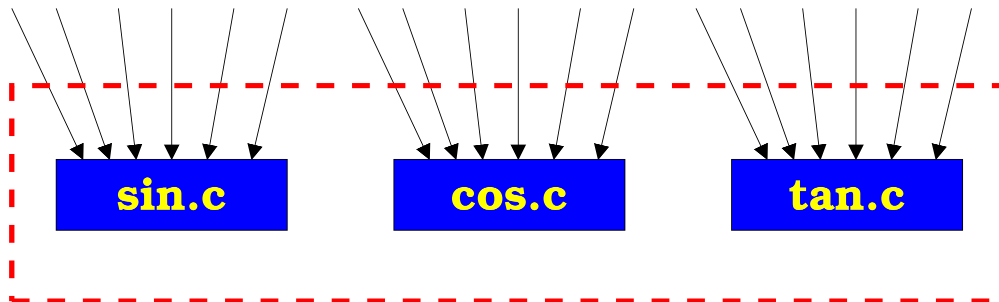


Figure 6.4: The leaf collection pattern.

Rationale

A pattern commonly observed in software systems is a set of resources that are not connected to each other, i.e. not dependent on each other, but they serve similar purposes, such as a set of drivers for various peripheral devices. These files are usually leaves in the system's graph, hence the name (this pattern is similar to the "shared neighbors" approach presented by Schwanke [SP89]).

From a comprehension point of view, it seems appropriate to study these resources together even though they do not interact with each other.

Instance naming

Resources that match the leaf collection pattern commonly reside in the same directory. If that is the case, the name of the directory can be used as before. Alternatively, a substring analysis as in [AL97] can be performed. Finally, if these do not work, the cluster can be named "leaf.ss".

Support library pattern

Description

This pattern groups resources of large in-degree into one subsystem. In other words, all resources that are heavily accessed by the rest of the system are grouped together.

Diagram

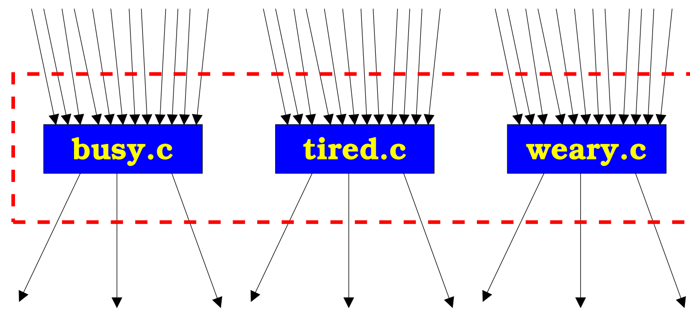


Figure 6.5: The support library pattern.

Rationale

Resources with large in-degree commonly obscure the structure of the software system (such resources are also referred to as "omnipresent nodes" [MOTU93]). Isolating them into their own subsystem usually makes discovering the structure of the rest of the system an easier task.

Instance naming

A substring analysis is again possible although it is rather unlikely to succeed this time, since these resources are not necessarily of similar nature. For this reason, we can name this subsystem "support.ss", a fairly descriptive and intuitive name.

Central dispatcher pattern

Description

This pattern is the dual of the support library pattern. It refers to resources of large out-degree. However, the number of such resources in a software system is limited (commonly only one). For this reason, the application of this pattern dictates that such resources are not considered until several clusters have already been formed. At that point, they can be reconsidered in conjunction with the already formed clusters.

Diagram

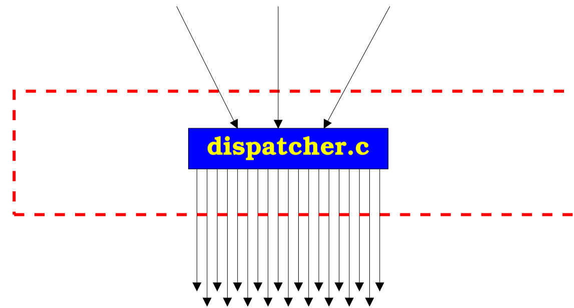


Figure 6.6: The Central dispatcher pattern.

Rationale

A common example of a resource with a large out-degree is a procedure (commonly called a driver) that calls many other procedures in order to execute certain parts of the system in sequence. For this reason, it is beneficial to wait for other subsystems to be formed before we include such a resource in a subsystem.

Also, the proliferation of edges originating from such a resource might obscure other patterns in the system's structure. This is why one initially disregards such resources and their outgoing edges.

Instance naming

Files with large out-degree are obviously integral parts of the software system. For this reason, their name followed by the suffix ".ss" can also be the name of the cluster that contains them.

Subgraph dominator pattern

Description

This pattern looks for a particular type of subgraph in the system's graph $G = (V, E)$. This subgraph must contain a node n_0 (called the "dominator node") and a set of nodes $N = n_i, i : 1..m$ (called the "dominated set") that have the following properties:

1. There exists a path from n_0 to every n_i.

2. For any node $v \in V$ such that there exists a path P from v to any $n_i \in N$, we have either $n_0 \in P$ or $v \in N$.

In simpler terms, one must go through node n_0 in order to get to any node $n_i \in N$.

This pattern can be extended so that instead of having a "dominator node" we have a "dominating set of nodes". However, this increases the complexity of the task of finding such subgraphs significantly, while it is not clear that the quality of the obtained clustering increases as well.

Diagram

See next page.

Rationale

This pattern attempts to discover collections of resources that cooperate in order to implement some specific functionality. The assumption is that the part of the software

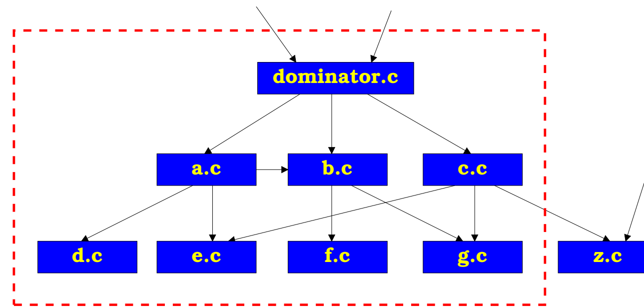


Figure 6.7: The subgraph dominator pattern. Note *z.c* is not included in the pattern instance, since there is a path to it that does not go through *dominator.c*

system that implements such functionality will have a single "point of entry", namely node n_0, and it will contain several files that interact only with each other. Experience has shown that this pattern emerges time and again in large software systems.

Instance naming

It is reasonable to assume that the name of the dominator node should provide an adequate description of the functionality implemented by the dominated set. In the worst case, it will be a familiar name that the developers of the software system can associate with. As a result, the name of the dominator node followed by the suffix ".ss" will be the name of any clusters created by the application of the subgraph dominator pattern.

The list of patterns presented above is by no means complete. Depending on the development philosophy adopted by different teams, or the specific nature of certain software systems, more patterns may be added.

An important point that should not be overlooked is that different patterns may suggest different clusterings for the same software system. Therefore, an algorithm must decide which patterns are appropriate, as well as how to combine information obtained

from different patterns in a meaningful and effective way.

In the next section, we will discuss how ACDC accomplishes this task. We will also present further details about the way our algorithm functions.

6.4 The ACDC algorithm

ACDC performs the task of clustering a software system's resources in two stages. In the first one, described in section 6.4.1, it creates a skeleton of the final decomposition by identifying subsystems using a pattern-driven approach. Depending on the pattern used, the subsystems are given appropriate names. In the second stage, described in section 6.4.2, ACDC completes the decomposition by using an extended version of the Orphan Adoption algorithm presented in section 5.2. Table 6.1 in the next page presents the two stages and the steps they contain in a compact form. In the following, we will look at both stages in more detail.

6.4.1 Stage 1: Skeleton construction

As noted in section 6.3, there is a variety of patterns that a pattern-driven clustering approach can attempt to identify. ACDC considers many of those patterns and performs the following steps in order of precedence:

1. *Cluster using the source file pattern.* It is often beneficial for the comprehension of a software system to cluster system resources in a way that follows the source file pattern. Therefore, ACDC clusters resources that reside in the same file together, and considers the source file as being an atomic entity for the following steps.

ACDC stages and steps		Name/Description
Stage 1		Skeleton construction using patterns
	Step 1	Cluster using the source file pattern
	Step 2	Cluster using the body-header pattern
	Step 3	Create the potential dominator list
	Step 4	Disregard omnipresent nodes
	Step 5	Cluster using the subgraph dominator pattern
	Step 6	Iterate the subgraph dominator pattern including omnipresent nodes
	Step 7	Create "support.ss" and dispatcher subsystems
Stage 2		Completion of clustering using Orphan Adoption
	Step 1	Apply the Orphan Adoption algorithm
	Step 2	Cluster orphans that were not adopted with high confidence using the leaf collection pattern

Table 6.1: The stages and steps of the ACDC algorithm.

2. *Cluster using the body-header pattern.* In terms of the conceptual structure of a software system, the interface and the implementation of a software module are parts of the same entity. For this reason, ACDC consolidates them into one cluster.

3. *Create the potential dominator list.* In this step, ACDC creates a list of all the nodes in the software system. We will call this list the *potential dominator list.* As the name indicates, this is a list of the nodes that will be examined to determine whether they qualify as the dominator node of a subsystem.

   The list is sorted in order of ascending out-degree, i.e. the first node in the list is the one with the smallest number of outgoing edges, and the last node in the list is the one with the largest number of outgoing edges (ties are resolved arbitrarily).

   The reason for the ascending order is the following: Nodes with smaller out-degree are more likely to induce smaller subgraph dominator pattern instances. By examining them first, when we get to nodes of larger out-degree, there is a better chance that the larger subgraph dominator pattern instances will contain already formed subsystems. This results in smaller cardinality for the final subsystems.

4. *Disregard omnipresent nodes.* As suggested in the description of the support library and central dispatcher patterns, nodes that have large in- or out-degrees tend to obscure the structure of the system. For this reason, at this step ACDC will remove from the potential dominator list, all nodes that have an in- or out-degree larger than 20.[1] These nodes will be placed in two other lists:

[1] The choice of this particular value (20) is guided by the fact that ACDC attempts to produce clusters of low cardinality (between 5 and 20). As a result, a node with an in- or out-degree larger than 20 should probably be placed in a different cluster than its neighbours.

   (a) *Support library list.* This list will contain all nodes with an in-degree larger than 20.

   (b) *Dispatcher list.* This list will contain all nodes with an out-degree larger than 20.

   Nodes that would qualify for both lists are entered in the support library list.

5. *Cluster using the subgraph dominator pattern.* This is is the main step of our algorithm. To begin this step, ACDC goes through each node n in the potential dominator list and examines whether n is the dominator node of a subsystem, according to the subgraph dominator pattern.

   When a non-empty dominated set is discovered, ACDC creates a subsystem containing the dominator node and the dominated set. Nodes in the dominated set are removed from the potential dominator list, unless the cardinality of the dominated set was larger than 20. In that case, they remain in the potential dominator node list, in an effort to further decompose the newly discovered subsystem.

   After all nodes have been examined, ACDC organizes the obtained subsystems, so that the containment hierarchy is a tree (it is possible that the aforementioned process has discovered two subsystems, one of which is a proper subset of the other). It might also remove some subsystems of small cardinality (the contents of a subsystem with a cardinality of 4 or less are moved to the higher level subsystem if that does not increase its cardinality above the threshold of 20).

6. *Iterate the subgraph dominator pattern including omnipresent nodes.* At this point, the top level view of the structure of the software system contains subsystems as

well as source files. ACDC induces dependencies between the top level entities and repeats the previous step, i.e. looking for subgraph dominator pattern instances. All top level nodes are considered, including the omnipresent nodes.

Any omnipresent nodes that are part of subsystems created during the execution of this step are removed from the support library or the dispatcher list.

7. *Create "support.ss" and dispatcher subsystems.* In this final step of the skeleton construction process, ACDC creates a subsystem called "support.ss". This subsystem will contain any resources that still remain in the support library list.

   Also, for each resource in the dispatcher list, ACDC creates a subsystem of cardinality 1 that contains the corresponding resource. For presentation purposes, it is possible that these resources are not contained in any subsystem, but they are rather part of the top level view of the system's structure. This seems to make sense, since they are obviously integral parts of the software system.

Even though a large proportion of the system's resources is already clustered at this point (see section 6.5.3), many files are still not assigned to a subsystem, since they did not fit any of the aforementioned patterns. The second stage of our algorithm attempts to complete the system's structure by assigning all remaining files to one of the existing subsystems.

6.4.2 Stage 2: Orphan Adoption

One of the tasks that the Orphan Adoption algorithm (described in section 5.2) accomplishes is placing resources that are not part of the current decomposition (these resources

are called orphans) in an appropriate subsystem. It operates on the assumption that most nodes have already been assigned to a cluster.

In our case, the skeleton decomposition that we constructed in Stage 1 serves as the existing structure, while the non-clustered files are the orphans. The only difference to the usual application of Orphan Adoption is that the number of orphans will probably be somewhat larger.

ACDC uses the Orphan Adoption algorithm to assign all resources to some subsystem. A small extension to the original algorithm is introduced at this point: Resources that cannot be assigned to any subsystem with a sufficient level of confidence, i.e. more than 50%, are examined to determine whether they follow the leaf collection pattern. If that is the case, ACDC creates a new subsystem that will contain these files.

6.4.3 Algorithm properties

The ACDC algorithm was designed to have the features that were presented in section 6.2 as essential for a software clustering algorithm. It also has certain other interesting qualities. Following is a list of the properties of ACDC:

- Effective cluster naming. Any time ACDC creates a new subsystem, it provides a name that is either intuitive or will be familiar to the people associated with the software system in question.

- Bounded cardinality. The cardinality of the obtained clusters is indeed bounded. Step 5 of the skeleton construction process (that creates the majority of the subsystems) attempts to create subsystems containing no more than 20 files. Larger subsystems are decomposed further if possible. The Orphan Adoption process may

add some extra contents into some subsystems, but that number is usually insignificant.

- Use of patterns. The pattern-driven nature of the algorithm is obvious. Several different patterns are considered, which should produce an output suitable for a program comprehension project.

- Nested and unbalanced clustering. A feature of the obtained decomposition that may not be immediately obvious is that it is nested and unbalanced. This usually happens because instances of the subgraph dominator pattern may contain other instances of the same pattern (figure 6.8 presents a simple example). We consider this to be a positive feature of our algorithm since it reflects the real-life situation of many software systems (certain parts are usually more complicated than others).

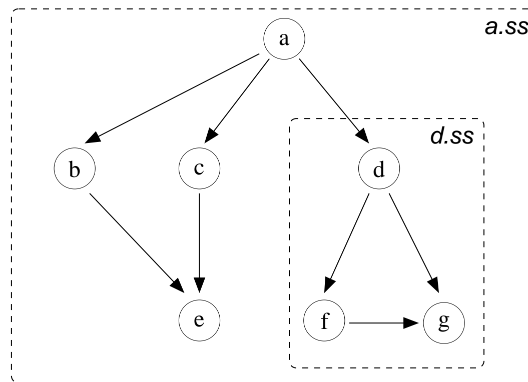


Figure 6.8: An example of nested subgraph dominator pattern instances. Node d will be examined before node a, and a cluster called "d.ss" containing nodes d, f, and g will be formed. When node a is examined, a cluster called "a.ss" will be formed containing nodes a, b, c, e, and the previously formed subsystem.

- Use of the directory structure pattern. Our algorithm makes rather limited use of the directory structure pattern. This is mainly due to our experience with several large industrial systems, where the directory structure did not seem to reflect the structure of the system, or where all files were stored in a single directory. However, this does not mean that the directory structure pattern is of no use. If it is considered an appropriate pattern for a given software system, its integration with our algorithm is easy (an extra step can be added between steps 2 and 3 of the skeleton construction process).

- Use of "magic numbers". Our algorithm utilizes a number of seemingly arbitrary values, such as the cardinality bounds of 5 and 20. These values are not fixed, and are actually parameters. However, choosing other reasonable values for these parameters has produced similar results to the ones presented in this chapter. For this reason, we chose to present the algorithm using fixed values.

In order to gauge the usefulness of our algorithm we conducted several experiments. The next section presents our results.

6.5 Validation of ACDC

No software clustering algorithm can claim to be successful, unless it has been tested on real-life systems. For this reason, we implemented ACDC as described in section 6.4, and tried it on the two large systems we have been experimenting with, TOBEY and Linux. In the following, we describe the experiments we conducted and present the obtained results.

All the experiments were run on a Sun Enterprise 450/4000 machine running Solaris 2.5.1.

6.5.1 Performance

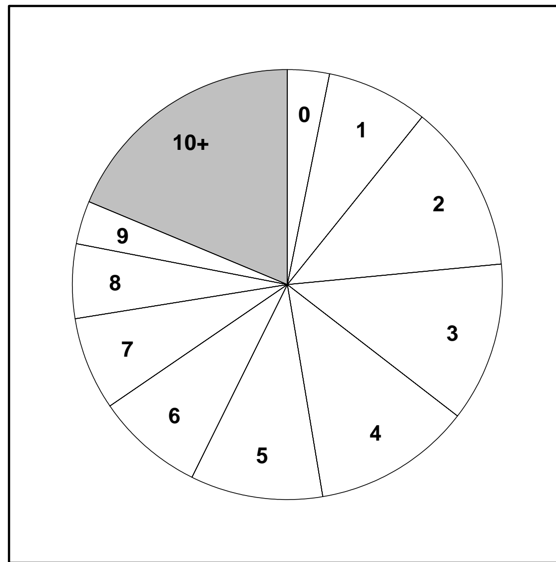
The execution speed of a clustering algorithm is not the most decisive factor for its evaluation. However, it is still desirable that the algorithm completes in a reasonable amount of time.

ACDC appears to perform well in that aspect. It required 54 seconds in order to cluster TOBEY. Linux, a somewhat larger system, required 84 seconds. Experience with other comparable systems indicates that ACDC can cluster industrial size systems in a reasonable amount of time.

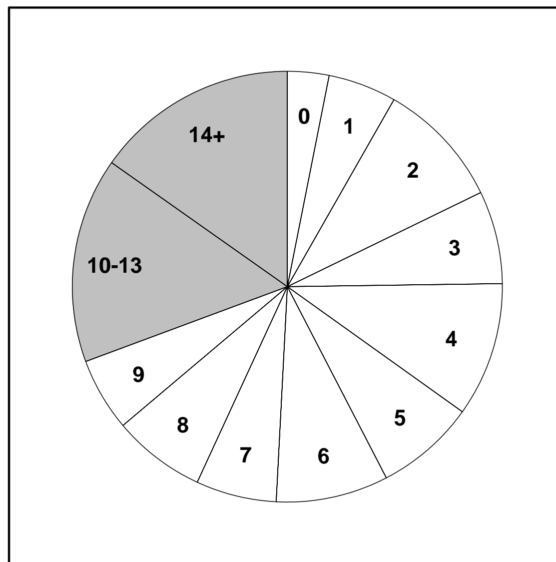
6.5.2 Stability

As discussed in chapter 4, it is desirable for a software clustering algorithm to be stable. Naturally, we were interested in measuring ACDC's stability. We applied the measure described in section 4.3 and obtained the results shown on the next page (figures 6.9a and 6.9b). As in section 4.4, the white slices correspond to experiments in which ACDC behaved in a stable fashion, while the shaded slices denote experiments in which ACDC came up with a decomposition that was significantly different than the original one. The label of each slice is the MoJo distance between the original decomposition and the one ACDC created for the randomly modified system.

Our experiment showed that ACDC behaved in a stable fashion in 81.3% of the repetitions when using TOBEY as input. In the case of Linux, the corresponding number



(a) $\Sigma\tau(ACDC,\text{TOBEY})$=81.3%



(b) $\Sigma\tau(ACDC,\text{LINUX})$=69.4%

Figure 6.9: Stability diagrams for ACDC with respect to TOBEY and Linux.

was 69.4%. Of the remaining 30.6% of the experiments, ACDC was only marginally unstable for more than half of them (indicated by the slice labeled 10-13). These results are definitely satisfactory, and indicate that ACDC is a stable algorithm. This makes it a good candidate for reverse engineering projects working with systems that are under development, since high stability predicts that clusterings of successive "snapshots" of the system will not be significantly different.

6.5.3 Skeleton size

An important condition that has to be satisfied in order to use the Orphan Adoption algorithm is that a large number of nodes must already belong to clusters. In our case, that means that the skeleton that is constructed in Stage 1 of our algorithm must contain a significant proportion of the system's resources.

Experiments showed that the TOBEY skeleton contained 604 of the 939 files of the system (a percentage of 64.3), while for Linux that number was 587 out of 955 (61.5%). Both of these values indicate that a well-formed structure is already in place before the Orphan Adoption process starts adding to it.

6.5.4 Quality

The goal of this experiment was to show that besides its comprehension qualities, ACDC is capable of producing meaningful decompositions, i.e. decompositions that resemble the actual structure of the software system. For this purpose, we employed the quality measure presented in section 3.6.3.

The quality measure values that we obtained were:

$$Q(ACDC, TOBEY) = \left(1 - \frac{326}{939}\right) \times 100\% = 65.3\%$$

$$Q(ACDC, Linux) = \left(1 - \frac{492}{955}\right) \times 100\% = 48.5\%$$

These values indicate that ACDC produces meaningful results as well as ones suitable for program understanding. The quality measure for TOBEY especially, is one of the higher ones an automatic clustering algorithm can hope to achieve. As a measure of comparison, table 6.2 presents the quality measure values for the six hierarchical clustering algorithms presented in chapter 4.

	SL65	SL75	SL90	CL65	CL75	CL90	ACDC
TOBEY	33.2%	20.4%	17.9%	43.8%	54.2%	60.4%	65.3%
Linux	17.9%	9.4%	4.1%	31.2%	33.4%	33.1%	48.5%

Table 6.2: Quality metric values for six hierarchical clustering algorithms and ACDC.

It is interesting to note the high quality metric value of algorithm CL90 when applied to TOBEY (60.4%). Careful inspection of all algorithms from CL85 to CL95 revealed that this is indeed the best quality metric value that a complete-linkage hierarchical clustering algorithm can achieve.

6.5.5 Observations

Table 6.3 presents a summary of the results of the experiments presented in this section.

An interesting observation that is apparent from the results of our experiment is that ACDC performs better with TOBEY rather than with Linux. The quality measure

	Performance	Stability	Skeleton size	Quality
TOBEY	54 sec	81.3%	604/939 (64.3%)	65.3%
Linux	84 sec	69.4%	587/955 (61.5%)	48.5%

Table 6.3: Summary of results.

values, as well as the stability results are strong indicators of that fact. The size of the TOBEY skeleton is also larger than the Linux one.

This phenomenon can probably be attributed to the nature of the two systems. TOBEY is a system developed by a relatively small-sized developing team that follows strict company regulations and a well-developed design. It is not surprising that its structure can be more easily discovered in an automatic way than Linux, a system developed by many people residing in various geographical locations and following different development philosophies.

Overall, the experiments we presented in this section suggest that ACDC is an effective software clustering algorithm. The quality of the clusterings it produces, along with its performance and stability make it a good candidate for reverse engineering projects.

In this chapter, we presented a software clustering approach that operates in a pattern-driven fashion by attempting to discover subsystems that resemble ones commonly observed in manually decomposed subsystems. We described an algorithm that implements this approach and explained why we think the clusterings it produces should help the process of understanding a software system. Finally, we presented experiments that showed the usefulness of our approach.

The next chapter summarizes the dissertation and indicates directions for further research.

Chapter 7

Conclusions

Industrial software systems are invariably large and complex. The sheer number of resources they contain, along with their intricate interconnections, makes the task of understanding their structure a hard one. The fact that parts of the source code may not have been maintained for years usually exacerbates the problem.

A major emphasis of software engineering research has been to assist developers in confronting the difficulties of dealing with large legacy software systems. Research in software clustering focuses on the development of techniques that attempt to alleviate the problem by decomposing large software systems into smaller, more manageable subsystems. Such techniques may assist a) system architects in detecting flaws in the concrete architecture of their system, b) newcomers to the software project in grasping its structure faster, or c) reverse engineers in understanding legacy code.

Our main contributions to the research area of software clustering are the realization that software clustering techniques must focus on creating decompositions that will be helpful to the software system's comprehension process, and the development of such an

approach based on subsystem patterns. Several other research challenges in the area of software clustering were also addressed in this dissertation.

The following section explains our research contributions in more detail.

7.1 Research contributions

1. Distance metric. We introduced a metric that measures distance between two partitions of the same set of data. This metric is useful in many scenaria, such as determining the stability of software clustering algorithms, or choosing appropriate values for any variables a parameterized approach might have. It can also be used in order to determine whether a clustering produced by a given algorithm comes close to the authoritative clustering produced by the software system's developers.

2. Stability. We studied the notion of stability of software clustering algorithms and we presented a measure that gauges it. Stability is an important property of a clustering algorithm that can be used as a means of comparison between different approaches. This dissertation presented the first extensive study on the stability of software clustering algorithms.

3. Incremental clustering. We developed an incremental clustering algorithm. Our Orphan Adoption algorithm can be used in order to keep the structure of an evolving software system up-to-date, as well as to indicate areas of the source code that might require restructuring.

4. Pattern-driven clustering. Our approach to clustering is based on the notion that software clustering efforts should be focused on helping understand the software sys-

tem. We developed a pattern-driven software clustering approach that subscribes to this philosophy, and presented an algorithm that exhibits certain features that are aimed towards this objective.

The following section offers suggestions on how to extend the work presented in this dissertation.

7.2 Suggestions for extending this research

Additional experimentation

Each of the contributions of this dissertation has been validated by experimenting with large software systems. However, there is definitely room for more experimentation. It would be interesting to try our approaches on a variety of software systems and determine whether we encounter similar behaviour or not.

MoJo complexity analysis

Section 3.4 mentions that determining the exact value of the MoJo metric is a hard task. However, it is not known exactly how hard this is. Work needs to be done in order to determine the computational complexity of this task.

Also, even though HAM (the heuristic algorithm for MoJo) performs well in practice, it would be interesting to determine its worst-case computational complexity.

Further stability study

Determining and categorizing the stability behaviour of various clustering algorithms is a task large enough for another dissertation. It would be very interesting if specific behaviour could be attributed to particular features of a clustering algorithm.

A first step towards this direction would be to experiment with more association coefficients and update rules. Also, instead of comparing different versions of hierarchical clustering algorithms when they use the same cut-point height, it would be interesting to compare their stability when they produce the same number of clusters.

Extended Orphan Adoption

The Orphan Adoption algorithm presented in this dissertation does not modify the number of clusters in the current decomposition, with the exception that a cluster that becomes empty is discarded.

However, it could be desirable to split clusters if they become too large, or merge them if they are small. It is also possible that a collection of orphans might have to be placed in their own subsystem. Work needs to be done to extend the Orphan Adoption algorithm in order to handle these cases.

Comprehension value of pattern-driven clustering

Our pattern-driven approach to software clustering was developed so that its results would be helpful to someone attempting to understand a given software system. It exhibits certain features, such as the effective naming of clusters, or their bounded cardinality, that we believe would aid the comprehension process.

However, work needs to be done in order to determine whether this is true in a real-life environment. An empirical study will have to be devised in order to determine whether developers actually benefit from the comprehension-driven features of our approach.

The next section provides some insights into new research opportunities that arise from our work.

7.3 Opportunities for further research

MoJo vs. Koschke-Eisenbarth

Approximately one year after the MoJo metric was published [TH99], another metric that measures similarity between partitions was presented by Koschke and Eisenbarth [KE00]. Their metric is loosely based on the Precision and Recall metrics. They do incorporate a formula that produces a single number that is representative of the two partitions' similarity.

It would be interesting to compare the two metrics and determine whether they exhibit similar behaviour or not, i.e. it may be possible that one metric decides that partition A is closer to partition B rather than partition C, while the other metric produces the opposite result.

Clustering based on dynamic dependencies

The work presented in this thesis is based on static dependencies between resources. It is not known whether our approaches will work as well if dynamic (run-time) dependencies were used instead. For example, it is likely that a different set of patterns might be

appropriate in this case. Further research is required in order to determine whether this is true.

Pattern-driven clustering for object-oriented systems

Object-oriented systems are usually not as old as their procedural counterparts. However, the recent years have seen an increase in the number of legacy object-oriented systems, and this trend is expected to continue. Automatic clustering for object-oriented systems might soon be a necessity.

Object-oriented patterns are already well documented. This makes it likely that a pattern-driven approach might be successful. However, this needs to be corroborated by experimental results. Furthermore, such an approach would have to resolve several issues, such as which patterns to employ and in what order.

General pattern-driven clustering

This thesis has focused on comprehension, and has shown that a pattern-driven approach is well-suited for this purpose. However, it is possible that such an approach might be effective in other contexts, such as when our goal is objectification or remodularization. Moreover, non-software clustering projects might benefit from such an approach. Defining the appropriate patterns might be the hardest obstacle in order to migrate this approach to other disciplines, but the opportunity for further research is definitely there.

7.4 Closing remarks

Our work contributes to the field of software engineering by introducing a novel approach to the software clustering problem, as well as addressing several related research challenges. We have presented experimental data that demonstrate that our techniques were effective in dealing with real industrial software systems.

We hope that this thesis will encourage researchers in software clustering to focus on comprehension and the development of techniques that will be helpful to developers of large software systems.

Appendix A

Example software systems

The majority of experiments presented in this thesis utilize two example software systems of comparable size but different development philosophy as input. The names of the two systems are TOBEY and Linux. This appendix will present them in detail.

A.1 TOBEY

This proprietary industrial system serves as the optimizing back end for a number of IBM compiler products. It has been continuously under development for more than a decade. Several parts of the system had not been maintained for years, something that could cause problems as the developing team was expanding. This fact made TOBEY an ideal legacy software system to work with.

TOBEY is written in PLIX, a proprietary procedural language. The version we worked with was comprised of 939 source files and approximately 250,000 lines of code. Due to some historical reasons, there was no directory structure for the source code, i.e.

all 939 source files resided in the same directory. Approximately half of the source files followed a rather loose naming convention (most of the time the two first letters of the source file name indicated the subsystem the file belongs in).

The "authoritative clustering" of TOBEY that we have utilized a number of times throughout this dissertation was obtained after a series of meetings with a seasoned developer, Ian McIntosh. The first meetings defined the main subsystems, while consequent meetings refined this structure. Figure A.1 presents the top level view of TOBEY's Software Bookshelf that contains the authoritative clustering. The layout of the subsystems is meant to reflect their order of execution.

Our research team extracted the dependencies between the various resources from TOBEY's source code. The dependency types we utilized were procedure calls and data references. The same dependencies were also included in TOBEY's Software Bookshelf. The visualizations presented to the developers were helpful for more reasons than just comprehension. In one case, a bug was uncovered by noticing a dependency that "should not be there". It turned out that two totally distinct parts of the system were using the same variable without knowledge of the other usage (PLIX allows the use of integer variables without declaration).

A.2 Linux

Linux is probably the most famous open-source system. It is being developed by many developers that reside in numerous geographical locations. It is an operating system for PCs that is similar to Unix.

Linux is written in C. We experimented with version 2.0.27a that is comprised of 955

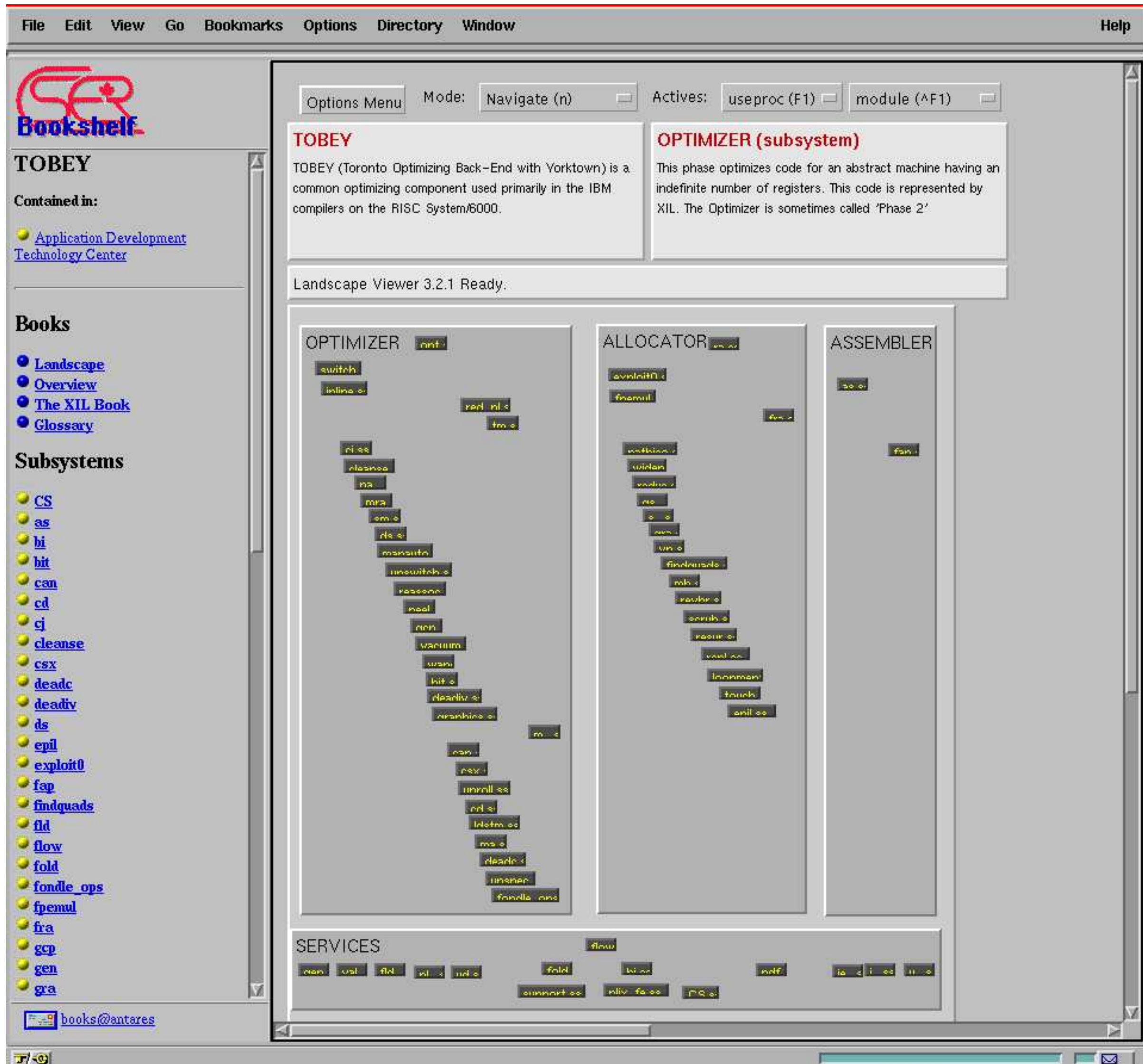


Figure A.1: The top level view of the TOBEY Software Bookshelf (dependencies have been omitted to avoid cluttering the picture.

source files and approximately 750,000 lines of code. There was a directory structure that included a /include directory that contained many header files. No naming convention is used for the source file names.

Our research team extracted the resource dependencies from the source code (these were again procedure calls and data references) and also created the "authoritative clustering" by consulting a variety of sources. This work has been published [BHB99] and the Software Bookshelf can be found online at the following address:

`http://swag.uwaterloo.ca/pbs/examples/linux/index.html`

The top level view of the Linux Software Bookshelf can be seen in figure A.2.

One of the reasons we chose Linux as an example system was the fact that it is an open-source system. As a result, it is available to everybody that would like to experiment with it. This could be a first step towards creating a suite of test systems that clustering approaches can use as benchmarks.

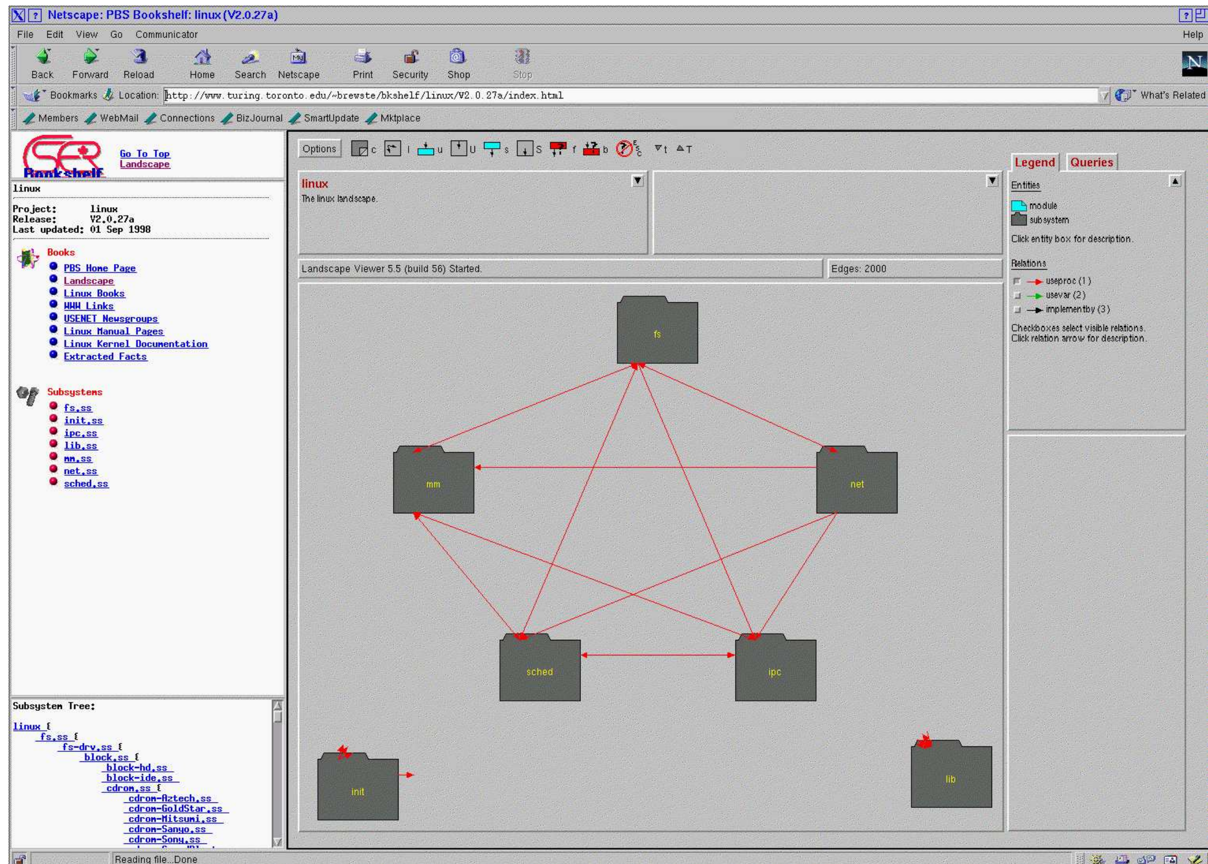


Figure A.2: The top level view of the Linux Software Bookshelf.

Appendix B

Software clustering toolkits

The two software clustering toolkits presented in this appendix are used throughout this thesis in order to compare our own software clustering approaches to existing ones. They were chosen for the following reasons:

- They are established approaches that have been successfully used on various software systems.
- They are publicly available.
- They can be run in batch mode.
- Their documentation was complete and helpful.

Despite these similarities however, they are quite different systems. In the following, we present them in detail.

B.1 HCAS

HCAS is a suite of programs one can use in order to implement practically any variation of a hierarchical clustering algorithm. It was developed by Nicolas Anquetil and was described in detail in [AL99]. HCAS is available for download from the following address:

`http://www.csi.uottawa.ca/~anquetil/Clusters/index.html`

In order to use this suite one starts with a description of the features of the entities to be clustered. HCAS utilizes only binary features, i.e. features that are either TRUE or FALSE for a given entity. These are input in a file of the following format (each entity is followed by a list of all features of value TRUE):

```
entity1 feat11 feat12 feat13
entity2 feat21 feat22
....
```

This file is then translated into an intermediate non-human-readable format called GdeM. The user can then issue the "cluster" command. This will create a dendrogram (see section 2.3.2) that the user has the option to cut at any desirable height. The "cluster" command has a lot of options allowing the user to specify a variety of similarity metrics and update rules. Finally, the output can be transformed from the GdeM format to Prolog facts.

The format of the facts that we extracted from each example software system is called RSF (for Rigi Standard Format) and is of the form:

```
dependency1 entity1 entity2
```

```
dependency2 entity3 entity4
....
```

As a result, in order to use the HCAS suite we transformed RSF to the HCAS format as follows: Each entity has all the entities it interacts with as features. Finally, in order to utilize the output of HCAS we had to write a program that translates the Prolog facts to RSF.

It should be noted that HCAS is a name we gave to this suite for convenience of reference. The creator of the suite did not provide a name for it.

B.2 Bunch

Bunch is a clustering tool that has been developed by Spiros Mancoridis and his research team at Drexel University. It has been presented in a number of publications [MMR+98, MMCG99] and is available for download from the following address:

`http://serg.mcs.drexel.edu/bunch/download/index.html`

Bunch is written in Java and can run in interactive, batch or distributed mode. It utilizes a metric called MQ that ranges from -1 to 1 and measures the quality of a given clustering in terms of its cohesion and coupling. In order to find a clustering that optimizes the value of the MQ metric, Bunch implements a number of different algorithms that are listed below:

1. *Exhaustive.* This algorithm examines all possible clusterings and selects the one that exhibits the largest MQ value. It is only usable for small graphs.

2. *NAHC*. This stands for Nearest-Ascend Hill Climbing algorithm. As the name suggests, it starts with a randomly-selected partition and examines neighbouring partitions. As soon as it finds one that exhibits a better MQ value it adopts it as the current partition and repeats the same step.

3. *SAHC*. This stands for Steepest-Ascend Hill Climbing algorithm. It also starts with a randomly-selected partition but it examines all neighbouring partitions. The one that exhibits the best MQ value is adopted and the same procedure is repeated.

4. *Genetic*. This algorithm uses ideas from genetic algorithms in order to find a partition that exhibits a good MQ value.

A number of different options accompanies each algorithm. The most important one is that of population size. This refers to the number of initial partitions that Bunch considers. The default value is 1 but any number can be chosen. Execution speed decreases linearly as the population size increases. Bunch also allows its user to specify "libraries" and "omnipresent modules" that should not be included in the clustering process.

Bunch accepts as input a file of the following format:

```
entity1 entity2
entity3 entity4
....
```

The above implies that entity1 depends on entity2, entity3 depends on entity4 etc., which meant that it was trivial to translate RSF to Bunch's input format.

Bunch can create output for Dotty, a graph visualization tool from AT&T Research, as well as provide textual output of the following format:

```
SS(1) = entity1, entity2, entity4

SS(2) = entity3, entity5, entity6

....
```

As a result, we only had to write a small program that translates Bunch's textual output to RSF in order to use Bunch for our experiments.

Bibliography

[AB87] Rade Adamov and Peter Baumann. *Literature Review on Software Metrics.* Institut für Informatik der Universität Zürich, October 1987.

[AL97] Nicolas Anquetil and Timothy Lethbridge. File clustering using naming conventions for legacy systems. In *Proceedings of CASCON 1997*, pages 184–195, November 1997.

[AL99] Nicolas Anquetil and Timothy Lethbridge. Experiments with clustering as a software remodularization method. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 235–255, October 1999.

[And73] M. R. Anderberg. *Cluster Analysis for Applications.* Academic Press Inc., 1973.

[BE81] L. A. Belady and C. J. Evangelisti. System partitioning and its measure. *Journal of Systems and Software*, 2:23–29, 1981.

[BH99] Ivan T. Bowman and R. C. Holt. Reconstructing ownership architectures to help understand software systems. In *Proceedings of the Seventh International Workshop on Program Comprehension*, May 1999.

[BHB99] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux as a case study: Its extracted software architecture. May 1999.

[BS91] Rodrigo A. Botafogo and Ben Schneiderman. Identifying aggregates in hypertext structures. In *Proceedings of Hypertext 91*, pages 63–74, 1991.

[CS90] Song C. Choi and Walt Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, pages 66–71, January 1990.

[CTH95] Ian H. Carmichael, Vassilios Tzerpos, and R.C. Holt. Design maintenance : Unexpected architectural interactions. *International Conference on Software Maintenance*, pages 134–137, October 1995.

[CW78] D. G. Corneil and M. E. Woodward. A comparison and evaluation of graph theoretical clustering techniques. *INFOR*, 16, 1978.

[DK76] Frank DeRemer and Hans H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, 2, 1976.

[DRN99] Stephane Ducasse, Tamar Richner, and Robb Nebbe. Type-check elimination: Two object-oriented reengineering patterns. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 157–166, October 1999.

[Eve93] Brian S. Everitt. *Cluster Analysis*. John Wiley & Sons, 1993.

[FHK+97] P.J. Finnigan, R.C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H.A. Müller, J. Mylopoulos, S.G. Perelgut, M. Stanley, and K. Wong. “The Software Bookshelf”. *IBM Systems Journal*, 1997.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability*. W. H. Freeman and co., 1979.

[GS93] David Garlan and Mary Shaw. An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering*, 1, 1993.

[Har75] John A. Hartigan. *Clustering Algorithms*. John Wiley & Sons, 1975.

[HB85] David H. Hutchens and Victor R. Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, 11(8):749–757, August 1985.

[HF98] R.C. Holt and Gary Farmaner. "The Portable Bookshelf". *http://www-turing.cs.toronto.edu/pbs*, 1998.

[HRY95] David R. Harris, Howard B. Reubenstein, and Alexander S. Yeh. Reverse engineering to the architectural level. In *International Conference on Software Engineering*, pages 186–195, April 1995.

[JD88] A. Jain and R. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.

[KE00] Rainer Koschke and Thomas Eisenbarth. A framework for experimental evaluation of clustering techniques. In *Proceedings of the Eighth International Workshop on Program Comprehension*, pages 201–210, June 2000.

[KL70] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal*, 49:291–307, 1970.

[Kon97] Kostas Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 44–55, October 1997.

[Lak97] Arun Lakhotia. A unified framework for software subsystem classification techniques. *Journal of Systems and Software*, pages 211–231, March 1997.

[LG95] A. Lakhotia and J. M. Gravley. Toward experimental evaluation of subsystem classification recovery techniques. In *Proceedings of the Second Working Conference on Reverse Engineering*, pages 262–269, July 1995.

[LS97] Christian Lindig and Gregor Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 19th International Conference on Software Engineering*, pages 349–359, May 1997.

[MMCG99] Spiros Mancoridis, B.S. Mitchell, Y. Chen, and E.R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society Press, 1999.

[MMM93] Ettore Merlo, Ian McAdam, and Renato De Mori. Source code informal information analysis using connectionist models. In *International Joint Conference on Artificial Intelligence*, pages 1339–1344, 1993.

[MMR+98] Spiros Mancoridis, B.S. Mitchell, C. Rorres, Y. Chen, and E.R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *IEEE proceedings of the 1998 International Workshop on Program Comprehension*. IEEE Computer Society Press, 1998.

[MOTU93] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5:181–204, December 1993.

[MU90] Hausi A. Müller and James S. Uhl. Composing subsystem structures using (k,2)-partite graphs. In *Conference on Software Maintenance*, pages 12–19, November 1990.

[MV99] Jonathan I. Maletic and Naveen Valluri. Automatic software clustering via latent semantic analysis. In *Proceedings of the Fourteenth IEEE International Conference on Automated Software Engineering*. IEEE Computer Society Press, 1999.

[Nei96] James M. Neighbors. Finding reusable software components in large systems. In *Proceedings of the Third Working Conference on Reverse Engineering*, pages 2–10. IEEE Computer Society Press, November 1996.

[Par72] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, December 1972.

[Pen92] David A. Penny. The software landscape: A visual formalism for programming-in-the-large. *Ph.D. Thesis, Department of Computer Science, University of Toronto*, 1992.

[RY81] Vijay V. Raghavan and C. T. Yu. A comparison of the stability characteristics of some graph theoretic clustering methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3, 1981.

[SAP89] Robert W. Schwanke, R.Z. Altucher, and Michael A. Platoff. Discovering, visualizing, and controlling software structure. In *International Workshop on Software Specification and Design*, pages 147–150. IEEE Computer Society Press, 1989.

[Sch91] Robert W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proceedings of the 13th International Conference on Software Engineering*, pages 83–92, May 1991.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives of an Emerging Discipline*. Prentice Hall, Englewood Cliffs, New Jersey, 1996.

[SP89] Robert W. Schwanke and Michael A. Platoff. Cross references are features. In *Second International Workshop on Software Configuration Management*, pages 86–95. ACM Press, 1989.

[TH99] Vassilios Tzerpos and R. C. Holt. Mojo: A distance metric for software clusterings. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 187–193, October 1999.

[Vas68] P. K. T. Vaswani. A technique for cluster emphasis and its application to automatic indexing. *Information Processing*, 68(2):1300–1303, 1968.

[vDK99] Arie van Deursen and Tobias Kuipers. Identifuing objects using cluster and concept analysis. In *Proceedings of the 21th International Conference on Software Engineering*, pages 246–255, May 1999.

[vL93] Gregor von Laszewski. A collection of graph partitioning algorithms. Technical Report SCCS 477, Northeast Parallel Architectures Center at Syracuse University, May 1993.

[Wig97] Theo A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 33–43. IEEE Computer Society Press, October 1997.

[ZSS92] Kaizhong Zhang, Rick Statman, and Dennis Shasha. On the editing distance between unordered labeled trees. *Information Processing Letters*, 42:133–139, 1992.

[Zup82] J. Zupan. *"Clustering of Large Data Sets"*. Research Studies Press, England, 1982.