METHODS FOR EVALUATING, SELECTING AND IMPROVING SOFTWARE CLUSTERING ALGORITHMS

MARK SHTERN

A DISSERTATION SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN COMPUTER SCIENCE AND ENGINEERING
YORK UNIVERSITY
TORONTO, ONTARIO
MAY 2010

METHODS FOR EVALUATING, SELECTING AND IMPROVING SOFTWARE CLUSTERING ALGORITHMS

by **Mark Shtern**

a dissertation submitted to the Faculty of Graduate Studies of York University in partial fulfilment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

© 2010

Permission has been granted to: a) YORK UNIVERSITY LIBRARIES to lend or sell copies of this dissertation in paper, microform or electronic formats, and b) LIBRARY AND ARCHIVES CANADA to reproduce, lend, distribute, or sell copies of this dissertation anywhere in the world in microform, paper or electronic formats *and* to authorise or procure the reproduction, loan, distribution or sale of copies of this dissertation anywhere in the world in microform, paper or electronic formats.

The author reserves other publication rights, and neither the dissertation nor extensive extracts for it may be printed or otherwise reproduced without the author's written permission.

METHODS FOR EVALUATING, SELECTING AND IMPROVING SOFTWARE CLUSTERING ALGORITHMS

by **Mark Shtern**

By virtue of submitting this document electronically, the author certifies that this is a true electronic equivalent of the copy of the dissertation approved by York University for the award of the degree. No alteration of the content has occurred and if there are any minor variations in formatting, they are as a result of the coversion to Adobe Acrobat format (or similar software application).

Examination Committee Members:

1. Jonathan Ostroff
2. Vassilios Tzerpos
3. Aijun An
4. Steven Wang
5. Sotirios Liaskos
6. Abdelwahab Hamou-Lhadj

Abstract

A common problem that the software industry has to face is the maintenance cost of industrial software systems. One of the main reasons for the high cost of maintenance is the inherent difficulty of understanding software systems that are large, complex, inconsistent (developed using mixed methodologies, have incomplete features) and integrated.

One of the approaches that has been developed to deal with 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. Many software clustering algorithms that attempt to automatically construct decompositions of large pieces of software have been presented in the literature. Different algorithms construct different decompositions. Therefore, it is important to have methods that evaluate the quality of such automatic decompositions.

This thesis presents approaches to a variety of problems associated with the evaluation of software clustering algorithms. Several methods that compare flat decompositions of software systems have been presented in the literature. We introduce the first methods for the evaluation of nested decompositions.

We also introduce a set of indicators that measure structure discrepancy between soft-

ware decompositions. The structure indicators can augment the evaluation picture provided by traditional comparison methods, such as MoJoFM and KE.

In addition, we introduce and quantify the notion of clustering algorithm comparability. It is based on the concept that algorithms with different objectives should not be directly compared. We also introduce a novel process for software clustering evaluation that utilizes multiple simulated authoritative decompositions.

The big selection of software clustering algorithms raises the question of how to select a software clustering algorithm that is best suited for a specific software system. In this thesis, we introduce a method for the selection of a software clustering algorithm for specific needs. The proposed algorithm selection method is based on a newly introduced formal description template for software clustering algorithms. Using the same template, we also introduce a method for software clustering algorithm improvement.

Acknowledgements

I would like to thank my supervisor Professor Vassilios Tzerpos for his inspiring interest in my study. He 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 department and examination committees. Their input has benefited this thesis in many ways.

Table of Contents

Abstract **iv**

Acknowledgements **vi**

Table of Contents **vii**

List of Tables **xii**

List of Figures **xiv**

1 Introduction **1**

1.1 The software clustering problem 4

1.2 Software clustering issues 5

1.3 Research contributions 8

1.3.1 Evaluation methods for nested software decompositions 11

1.3.2 Clustering Evaluation Using Structure Indicators 13

1.3.3 Comparability of Software Clustering Algorithms 14

1.3.4 MoJoFm vs KE 15

1.3.5 Evaluation of clustering algorithms by utilizing multiple simulated authoritative decompositions . . . 16
1.3.6 Methods for Selecting and Improving Software Clustering Algorithms 17
1.4 Dissertation outline . . . 19

2 Background **22**
2.1 Software Clustering . . . 23
2.1.1 Fact Extraction . . . 24
2.1.2 Filtering . . . 33
2.1.3 Similarity computation . . . 35
2.1.4 Cluster Creation . . . 41
2.1.5 Results Visualization . . . 51
2.1.6 Feedback . . . 52
2.2 Definitions . . . 53
2.3 Evaluation of Clustering Algorithms . . . 55
2.3.1 Evaluation based on authoritative decomposition . . . 56
2.3.2 Evaluation methods not based on an authoritative decomposition . . 61
2.4 Conclusions . . . 65

3 Factbase and Decomposition Generators **73**
3.1 Simulated Factbase Generator . . . 75
3.2 Decomposition Generators . . . 78
3.2.1 Flat Decomposition Generators . . . 79

3.2.2 Nested Decomposition Generator 80
3.3 Decomposition Generator Properties 83
3.3.1 Uniqueness 83
3.3.2 Randomness 85
3.4 Applications 86
3.4.1 A simple clustering evaluation method 87
3.4.2 Properties of Bunch's MQ function 89
3.4.3 Congruity of Clustering Evaluation Measures 91

4 Evaluation of nested decompositions **94**

4.1 Flat vs. nested decompositions 95
4.2 The Evaluation of Nested Decompositions (END) framework 100
4.2.1 END framework 100
4.2.2 Experiments 103
4.3 UpMoJo Motivation 113
4.4 UpMoJo distance 115
4.4.1 The UpMoJo Algorithm 115
4.4.2 UpMoJo Discussion 121
4.4.3 Experiments 126

5 Structure Indicators **130**

5.1 Structure Evaluation 131
5.2 Structure Indicators 136

5.3 Experiments . . . 140
5.3.1 ELF evaluation . . . 141
5.3.2 Segment Threshold Evaluation . . . 146

6 Algorithm Comparability 148

6.1 Clustering Algorithm Comparability . . . 150
6.1.1 Quantifying Comparability . . . 150
6.1.2 Experiments . . . 152
6.2 A new clustering evaluation process . . . 157
6.2.1 Simulated Authoritative Decompositions . . . 158
6.2.2 LimSim . . . 161
6.2.3 LimSim Experiments . . . 163
6.3 MoJoFM and KE . . . 164
6.3.1 Structure reduction . . . 165
6.3.2 Comparability reduction . . . 166

7 Methods for selecting and improving software clustering algorithms 169

7.1 Describing software clustering algorithms . . . 171
7.1.1 Algorithm Selection . . . 173
7.1.2 Algorithm Improvement . . . 175
7.2 Case Studies . . . 178
7.2.1 Algorithm Rejection . . . 178
7.2.2 Body - Header Processing . . . 180

7.2.3 Component - Library Processing 183

7.2.4 Improving Bunch 186

7.3 Discussion 188

8 Conclusions **191**

8.1 Research contributions 192

8.2 Future Research Directions 195

8.3 Closing remarks 199

A Examples of algorithm templates **200**

A.1 ACDC Algorithm 200

A.2 Bunch 204

A.3 Limbo 206

A.4 Agglomerative Clustering Algorithms 208

Bibliography **210**

List of Tables

1.1 Mapping between issues and contributions. 10

2.1 Data Table Example. 36

2.2 Categorical Attribute Presented as Binary Data. 36

2.3 Categorical Attribute in compact form. 37

2.4 Examples of binary resemblance coefficient. 38

3.1 The relation between algorithm version, inputs, and output decomposition type. 80

3.2 Results for the simple clustering evaluation method. 89

3.3 Evaluation measure values for all decomposition pairs. 91

4.1 Nested decompositions used in the experiments. 104

4.2 Summary of Comparing Nested Decompositions (C = compact, D = detailed, E = END). 105

4.3 Formulas for all weighting schemes. 109

4.4 Experimental results with various weighting schemes. 111

4.5 MinUpMoJo and UpMoJo results. 126

4.6 Congruity metric values for several experimental setups. 129

5.1 MoJoFM and KE values for decompositions `F` and `N`. 135

5.2 Decompositions used in the experiments. 142

5.3 Comparison results for flat decompositions. For E and L, the total number of clusters is shown in parentheses. 143

5.4 Comparison results for nested decompositions. For E and L, the total number of clusters is shown in parentheses. 145

5.5 Comparison results for flat decompositions using a segment threshold of 0.7.146

6.1 Pair-wise comparability results for the selected clustering algorithms. 153

6.2 Comparability of ACDC to Hierarchical Algorithms. 154

6.3 Comparability results for constant update rule function and varying similarity metric (Jaccard and Sorensen–Dice). 155

6.4 Comparability results for constant similarity metric and varying update rule function. 156

6.5 LimSim results for Linux and TOBEY using MoJoFM as similarity. 164

7.1 Number of misplaced header files in automatically created decompositions 181

7.2 MoJoFM values for ACDC and Bunch before and after modification 182

7.3 MoJoFM values for ACDC and Bunch before and after modification 186

List of Figures

2.1 An example of a flat decomposition. 54

2.2 An example of a nested decomposition. 54

3.1 Pseudocode for the simulated nested decomposition generation process. . . 82

3.2 An example of two isomorphic nested decompositions. 84

3.3 A rooted tree MDG. 87

3.4 The decomposition that should correspond to the MDG in Figure 3.3. 88

4.1 Conversion of a nested decomposition to a compact flat decomposition. . . 96

4.2 The detailed flat form of the decomposition in Figure 4.1(a). 97

4.3 Limitation of compact flat decompositions. 98

4.4 Limitation of detailed flat decompositions. 99

4.5 Flat Decomposition P_2 for the decomposition in Figure 4.3b. 101

4.6 Comparing END to various standalone approaches. 106

4.7 Weighting scheme results. 110

4.8 Decompositions C and D. 114

4.9 Decompositions A and B. 114

4.10 An example of a hierarchical mistake ignored by END. 115
4.11 The containment tree of an automatic decomposition A. 116
4.12 The containment tree of the authoritative decomposition B. 116
4.13 The flat decomposition of nested decomposition A. 118
4.14 The flat decomposition of nested decomposition B. 118
4.15 Decomposition A after the transformation indicated by MoJo. 119
4.16 The flat decomposition of subsystem A45. 120
4.17 The flat decomposition of subsystem B2. 120
4.18 The final containment tree for decomposition A. 121
4.19 Pseudocode for the UpMoJo algorithm. 122
4.20 Example of 1 Up operation. 123
4.21 An example of an implicit Down operation. 124
4.22 The flat decompositions of the decompositions in Figure 4.21. 125
4.23 Limitation of MinUpMoJo distance. 127

5.1 Authoritative decomposition A. 132
5.2 Automatic decomposition B. 132
5.3 Automatic decomposition C. 133
5.4 Authoritative decomposition A. 134
5.5 Automatic decomposition F. 134
5.6 Automatic decomposition N. 134
5.7 Decomposition A of a software system. 136

5.8 Decomposition T of the software system. 137

5.9 The fragmentation distribution graph for our example. 139

5.10 Fragmentation Distribution Graphs. 144

6.1 Example of merging modules A and B. 160

1 Introduction

The most important goal of any software company is to deliver projects on-time and on-budget. Today, software projects are increasingly complicated due to the extensive number of functional/non-functional requirements, as well as the fact that various components from different vendors need to be integrated. As a result, meeting the aforementioned goals becomes harder and harder.

Software engineering research has developed methodologies that intend to guide software development teams towards meeting these goals. These methodologies outline the best development practices. Ideally, a software project starts with requirements gathering followed by specification. Then, the project reaches its major milestones with system design and implementation, quality assurance, and delivery. The last phase is operation and maintenance.

Documenting the software system is a crucial part of the software development process. The architecture's documentation is supposed to be traceable to the system's formal requirements, and to be consistent with all design and quality assurance documents. Design documents should be followed during implementation stages. During a system's evolution, all relevant documents should be updated and followed.

At the same time, software engineering guidelines explain the importance of selecting appropriate technologies, architecture styles, and third party software components. In general, new solutions should be a composition of reusable software components and strive to be generic.

Unfortunately, in real life, a development team may not be able to follow these guidelines due to various factors such as:

1. Time-lines of product development are too short
2. Product requirements are frequently modified
3. The product specification may be contradictory
4. Shortage of human resources
5. Human factors such as laziness, design/implementation mistakes, conflicts between different team players, people leaving the company, different development styles of various developers, different technology backgrounds etc.
6. Developers copy/paste legacy code from the public domain introducing large segments of code they do not necessarily understand
7. The system architecture guidelines are not followed by the developers
8. Decisions on vendors or technology are tied to commercial marketing requirements

Typical consequences of these factors include:

1. Developers are afraid to modify/delete source code developed by other members of the project. Therefore, they copy and paste existing code instead of refactoring or reorganizing modules
2. The documentation and the software system are not synchronized
3. The developers patch bugs (try to address the symptoms of a bug) instead of fixing them (getting to the real source of the problem)
4. The development team does not understand the whole system

The maintenance cost of such products is extremely high. It has been widely acknowledged that maintaining existing software accounts for as much as 60-80% of a software's total cost [LZN04]. The complexity of software products grows significantly after new features are added or new bugs are fixed. As a result, in-house developed systems become legacy systems. Moreover, companies start to lose money on the support of such products.

The software engineering communities have identified this problem and are developing different approaches to help companies overcome the shortage of knowledge about their software systems and to improve software quality. A subset of these approaches are based on the utilization of software clustering algorithms.

Software clustering algorithms identify groups of entities, such as classes or source files, whose members are in some way related. In other words, the goal of software clustering is to decompose large software systems into smaller, more manageable subsystems

that are easier to understand. Many different algorithms that create software decompositions automatically have been presented the literature[AT03, CS90, HB85, MMCG99, MU90, TH00a]. Several tools have been developed that have demonstrated promising results in industrial projects [Tze01, LZN04, AATW07].

1.1 The software clustering problem

Software clustering algorithms have been actively developed during the last two decades. 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 Conference on Program Comprehension.

The published studies can be classified in the following categories:

- *New Algorithms*. The studies that belong to this category present new software clustering algorithms or improve on existing ones [MOTU93, CS90, AL97, LS97, SAP89, MV99].

- *Applications*. The goal of these studies is to apply software clustering algorithms to different software engineering problems. The first application of software clustering was the decomposition of a software system into subsystems. Since then, software clustering methods have been applied in different software engineering contexts, such as reflexion analysis, software evolution and information recovery [MNS95,

ZKS04, Sch91, CSOT08].

- *Evaluation*. Evaluation studies present techniques for the evaluation of the effectiveness of software clustering algorithms. This is typically done by comparing the decompositions produced by clustering algorithms to *authoritative decompositions* created by system experts. The main benefits of evaluation methodology are:

  - Software clustering effectiveness is usually evaluated based on case studies that are often subjective. Evaluation methodologies are more objective since they typically do not require human input.

  - Evaluation helps discover the strengths and weaknesses of the various software clustering algorithms. This allows the development of better algorithms through addressing the discovered weaknesses.

  - Evaluation can help indicate the types of systems that are suitable for a particular algorithm.

  A number of evaluation techniques have been developed [TH99, ST04, MM01b, AL99, KE00].

Since the evaluation of software clustering methods is an important factor for the successful future development of software clustering methods, the majority of this thesis focuses on problems related to software evaluation. In the next section, we discuss several issues with the current state of the art for software clustering.

1.2 Software clustering issues

Research on software clustering has been active for more than twenty years. Despite the progress that has been made, there is still a lot of room for improvement. Several of the issues that researchers on software clustering are facing are discussed in this section.

- *Insufficient evaluation*. Perhaps the most important issue is that of the evaluation of an approach's usefulness. Most researchers attempt to validate their work by applying it to a small number of software systems. Therefore, it is not possible to generalize the evaluation results to other software systems.

- *Limited number of authoritative decompositions*. Only a small number of reference systems have available authoritative decompositions. Moreover, some reference systems are outdated, their source code no longer supported. It is not feasible to assess the effectiveness of a software clustering algorithm using only a handful of data points.

- *Result comparison*. This brings us naturally to another important problem, that of comparing the results of various approaches. 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 architect 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 in her schedule.

- *Interpretation of evaluation results*. Existing measures provide a single number that is representative of the quality of the automatic decomposition. However, there could be many underlying reasons that result in higher or lower quality, since there are several ways in which an automatic decomposition can differ from an authoritative one. A better understanding of these partial differences would be beneficial to software maintenance activities.

- *Evaluation of nested decompositions*. The evaluation of results obtained from software clustering algorithms has attracted the attention of many reverse engineering researchers. Several methods that compare flat decompositions of software systems have been presented in the literature. However, software clustering algorithms often produce nested decompositions, in which subsystems may be subdivided into smaller subsystems. Converting nested decompositions to flat ones in order to compare them may remove significant information.

- *Algorithm comparability*. A property of software clustering algorithms that has not attracted much attention is their comparability. In other words, more effort needs to be put into understanding whether it is meaningful to compare the results of different algorithms, since different algorithms have different clustering objectives. Comparability between a newly introduced and a known algorithm can be important for understanding the properties of the new algorithm.

- *Correlation of evaluation measures*. Each clustering publication uses a different set of evaluation measures from the several measurements that have been presented in the

literature. Consequently, it is not possible to compare evaluation results from different studies. To overcome this problem, the evaluation metrics need to be compared to each other in order to determine if they are correlated or not.

- *Formal description template*. Up to now, there is no formal description template that allows to formally specify a software clustering methodology. The adoption of such a template by software clustering researchers should lead to better communication in this research field and the development of more effective software clustering approaches.

- *Clustering algorithm selection*. Finally, several software clustering algorithms have been proposed in the literature, each with its own strengths and weaknesses. Most of these algorithms have been applied to particular software systems with considerable success. However, the question of how to select a software clustering algorithm that is best suited for a specific software system remains unanswered.

In the next section, we present the contributions of this thesis that address the aforementioned issues.

1.3 Research contributions

The main research contributions of this thesis are:

1. *The introduction of two methods that measure distance between two nested decompositions of the same software system*. A number of approaches that measure (dis)similarity

between decompositions of the same system have been presented in the literature [KE00, MM01a, TH99]. However, all existing methods operate only on flat decompositions. Our methods are the only methods capable of evaluating nested decompositions.

2. *The introduction of a set of indicators that measure structural differences between two decompositions of the same software system*. Existing evaluation measures provide a single number that is representative of the quality of the automatic decomposition. This is insufficient for understanding the underlying reasons for the differences between the two decompositions. The proposed set of indicators augments the evaluation picture provided by traditional comparison methods.

3. *The introduction of the notion of comparability of software algorithms and the presentation of a measure that gauges it*. The typical way to evaluate a software clustering algorithm is to compare its output to a single authoritative decomposition. If the authoritative decomposition was constructed with a different point of view in mind than the algorithm, the evaluation results will probably be biased against the algorithm. This thesis introduces and quantifies the notion of software clustering algorithm comparability. It helps to identify algorithms that should not be directly compared due to having different clustering objectives.

4. *The investigation of possible reasons for the discrepancies between two of the more popular evaluation methods, MoJoFM[WT04b] and KE[KE00]*. This is the first comprehensive study for the comparison of the two evaluation measures. The ideas presented in

this study can be applied to the comparison of any two evaluation measures for software clustering algorithms.

5. *The evaluation of clustering algorithms by utilizing multiple simulated authoritative decompositions*. We introduce a novel approach that allows to evaluate a software clustering algorithm on a large number of simulated authoritative decompositions.

6. *The introduction of processes for the selection and improvement of software clustering algorithms*. The question of how to select a software clustering algorithm that is best suited for a specific software system remains unanswered. In this thesis, we present a method for the selection of the best suited algorithm for a specific reverse engineering task. Also, we introduce a method for the improvement of existing software algorithms. Our approach allows the evaluation of the new algorithm in earlier stages, before it is fully implemented and finalized.

Several of these contributions have already been published in conference proceedings [ST04, ST07, ST09a, ST09b].

The remainder of this section elaborates on these six contributions, while in the next section we present an outline of the dissertation. Table 1.1 presents which contribution addresses each of the issues presented in Section 1.2.

1.3.1 Evaluation methods for nested software decompositions

We have developed the END framework and the UpMoJo distance measure. Both methods allow a researcher to compare nested decompositions of large software systems with-

Issue	**Contribution ID**
Insufficient evaluation	5
Limited number of authoritative decompositions	5
Result comparison	1,2
Interpretation of evaluation results	2
Evaluation of nested decompositions	1
Algorithm comparability	3
Correlation of evaluation measures	4
Formal description template	6
Clustering algorithm selection	6

Table 1.1: Mapping between issues and contributions.

out having to lose information by transforming them to flat ones first.

END framework

The END framework [ST04] is able to compare two nested decompositions without any information loss by converting them into vectors of flat decompositions and applying existing comparison methods to corresponding elements of these vectors. The overall similarity between the two nested decompositions is computed based on the similarity vector generated from comparing the decomposition vectors.

The main benefit of the END framework is that it allows users to utilize existing flat evaluation measures that they might be already familiar with. However, the END framework does not specify the function that will be used to convert the similarity vector to a number. While this makes the framework more flexible, it also increases the responsibility of the user. In practice, it is often difficult to differentiate between different weighting functions, as the framework does not provide any guidelines as to which functions are best suited in which situations. This can result in different users getting different results while comparing the same nested decompositions.

Moreover, one of the properties of the END framework is that a misplaced object closer to the root of the containment tree results in a larger penalty than if the misplaced object were deeper in the hierarchy. The END framework also does not penalize for some trivial hierarchical mistakes.

The UpMoJo [ST07] measure was developed to address these issues.

UpMoJo distance

UpMoJo distance provides a comparison method that is solely dependent on the two decompositions. It is an extension of MoJo distance [TH99]. UpMoJo calculates the number of Up, Move, and Join operations required to transform one nested decomposition to another using a pre-determined process. The Move and Join operation are the same operations as defined in the MoJo method. The Up operation moves an object or a set of objects that initially reside in the same subsystem S to the subsystem that directly contains S.

The UpMoJo transformation process traverses the automatic nested decomposition and transforms its every sub-tree into a corresponding authoritative sub-tree. To do this, UpMoJo performs the following steps:

1. Perform Up operations for all entities that require it so that the first levels of the two corresponding sub-trees contain the same entities.

2. Apply the Move and Join operations that MoJo would require for the first levels of the corresponding sub-trees.

3. Repeat the same process for the newly created sub-trees.

1.3.2 Clustering Evaluation Using Structure Indicators

We introduce three indicators that can be used to evaluate the structure of an automatic decomposition [ST09b]. They are all based on the intuition that two decompositions have similar structure when a software engineer gets the same picture about the software sys-

tem by reading either of these decompositions. The indicators are presented in the context of flat decompositions. If necessary, one can use the END framework to apply any of these indicators to a nested decomposition as well.

Following is an informal presentation of these structure indicators:

Extraneous Cluster Indicator (E): The number of clusters in the automatic decomposition that do not have corresponding clusters in the authoritative decomposition.

Lost Information Indicator (L): The number of clusters in the authoritative decomposition that do not have corresponding clusters in the automatic decomposition.

Fragmentation Indicator (F): Let us denote by S the number of clusters in the automatic decomposition that have corresponding clusters in the authoritative decomposition. The fragmentation indicator is defined as:

$$\begin{cases} \frac{S}{|A|-L} & \text{when} \quad |A| > L \\ 0 & \text{when} \quad |A| = L \end{cases}$$

where $|A|$ is the number of clusters in the authoritative decomposition.

These indicators also allow researchers to investigate the differences between existing evaluation approaches in a reduced search space. The structure indicators allowed the comparison of MoJoFM and KE in a reduced search space, i.e. when the decompositions involved have the same structure.

1.3.3 Comparability of Software Clustering Algorithms

We introduce and quantify the notion of software clustering algorithm comparability [ST10]. Comparable algorithms use the same or similar clustering objectives, produce similar results, and can therefore be evaluated against the same authoritative decomposition. Non-comparable algorithms use different clustering objectives, produce significantly different results, and cannot be evaluated by direct comparison to an authoritative decomposition.

We consider two algorithms to be non-comparable if they produce significantly different results across a variety of software systems. More precisely, we define that two algorithms are non-comparable if in the majority of cases they produce results that are more different than similar, i.e. the similarity is less than 50%.

In order to calculate comparability, the method requires a large number of dependency graphs extracted from software systems. Since this is impractical to do, we developed an algorithm to produce simulated dependency graphs.

We determined the comparability of the following clustering algorithms: ACDC [TH00a], Bunch [MMR+98], LIMBO [AT05], and several hierarchical clustering algorithms. We have shown that most major algorithms are non-comparable.

1.3.4 MoJoFm vs KE

One of the open questions in software clustering is whether clustering similarity measures, such as MoJoFM and KE, behave in a congruent fashion. The stated goal of both

measures is to quantify the quality of an automatic decomposition with respect to an authoritative one. However, they attempt to measure quality in different ways often resulting in contradictory results [AATW07, AT05].

To address this issue, we introduce a congruity measure [ST07] that measures correlation between two evaluation metrics M and U. The congruity metric generates N random pairs of decompositions of the same software system. Then it applies both M and U to each pair and obtains two different values m_i and u_i. We arrange the pairs of decompositions in such a way so that for $1 \leq i \leq N-1$ we have $m_i \leq m_{i+1}$. The value of the congruity metric is the number of distinct values of i for which $u_i > u_{i+1}$. Values for this metric range from 0 to $N-1$. A value of 0 means that both comparison methods rank all pairs in exactly the same order, while a value of $N-1$ probably indicates that we are comparing a distance measure to a similarity measure. Values significantly removed from both 0 and $N-1$ indicate important differences between the two comparison methods.

Using the congruity measure, we conclude that MoJoFM and KE are uncorrelated measures. We attempted to remove the effect that structure discrepancies between the automatic and authoritative decomposition may have in the measures' assessment. The structure indicators presented in Section 1.3.2 allowed us to compare MoJoFM and KE in a reduced search space, i.e. when the decompositions involved have the same structure. The results indicate a lack of correlation between MoJoFM and KE even when they are comparing decompositions with the same structure.

The notion of comparability defined in Section 1.3.3 allowed us to investigate this issue from a different angle: Is the behaviour of the two measures more congruent when

evaluating comparable algorithms? Once again, experimental results indicate a clear lack of correlation between MoJoFM and KE even when they are comparing decompositions produced by comparable algorithms.

A possible explanation for this rather surprising result (given that the goal of the two measures is the same) is the way they penalize for misplaced entities [ST09b].

1.3.5 Evaluation of clustering algorithms by utilizing multiple simulated authoritative decompositions

We present a novel method for the evaluation of software clustering algorithms, called LimSim. It supports evaluation of software clustering algorithms based on a large number of simulated software systems with available authoritative decompositions. The method is presented informally below:

We start with a real software system, such as Linux, for which there is a module dependency graph (MDG) and an authoritative decomposition. Next, we apply a series of small modifications to the MDG with corresponding modifications for the authoritative decomposition. Each modification is designed in such a way as to ensure that the resulting decomposition is indeed authoritative for the resulting MDG.

After a large number of randomly selected modifications has been applied, we have an MDG that is significantly different than the original one for which an authoritative decomposition exists. By repeating this randomized process many times, we can obtain a large population of simulated software systems with available authoritative decompositions.

1.3.6 Methods for Selecting and Improving Software Clustering Algorithms

We propose an algorithm for the selection of an appropriate software clustering method that is able to address specific reverse engineering needs and suitable for a targeted software system [ST09a]. The proposed selection method has three stages:

1. **Construct prototype decomposition**. A prototype decomposition is a crude, high-level decomposition of the software system that can be constructed with a small amount of effort utilizing expert knowledge, existing documentation, and the reverse engineer's experience.

2. **Apply algorithm restrictions**. The formal description of any candidate software clustering algorithm will contain a number of algorithm restrictions. If the given software system or its prototype decomposition match any of these restrictions, then the algorithm is rejected.

3. **Match decomposition properties**. A candidate software clustering algorithm is selected if the prototype decomposition can be produced by the algorithm. This means that the decomposition properties of the candidate algorithm must be consistent with the prototype decomposition.

Furthermore, we introduce a method for the improvement of existing software algorithms. The proposed conceptual work flow has the following phases:

- **Formulate an idea**. A software clustering researcher conceives of a novel idea based on her experience and domain knowledge.

- **Verify the idea**. The purpose of this stage is to determine whether available authoritative decompositions of known software systems have properties that reflect the clustering idea being considered.

- **Implement the new approach**. The full implementation of the new clustering idea takes place during this stage.

- **Evaluate the new approach**. The effectiveness of the new clustering idea can be evaluated by comparing its results to the results of existing approaches by using established evaluation criteria, such as MoJoFM or KE.

- **Document the approach**. The last stage of our proposed work flow requires that a formal description of the new clustering approach is created.

We have carried out several examples of applying these methods for algorithm selection and algorithm improvement problems.

Finally, we propose a consistent format for the description of software clustering algorithms [ST09a], not unlike the one used for design patterns [GHJV95]. The proposed format was used to describe several known software clustering algorithms. Their descriptions are available online [wik] as part of a community repository for software clustering methodology.

1.4 Dissertation outline

- **Chapter 1 - Introduction**

  This chapter introduces the software clustering problem and identifies some of the open questions in the area. The chapter concludes with an outline of the dissertation.

- **Chapter 2 - Background**

  This chapter presents the state of the art of software clustering methodology. It also discusses important research challenges in the area. The chapter concludes with a description of the software systems we experiment within this thesis.

- **Chapter 3 - Factbase and Decomposition Generators**

  This chapter presents methods for the generation of simulated software decompositions and factbases. In addition, we introduce the congruity metric that measures whether two evaluation metrics are correlated.

- **Chapter 4 - Evaluation of nested decompositions**

  This chapter outlines the rationale behind UpMoJo distance and the END framework, describes the details of our algorithms, and presents experiments we have conducted.

- **Chapter 5 - Structure Indicators**

  This chapter introduces a novel set of structure indicators that can augment the

evaluation picture provided by traditional comparison methods, such as MoJoFM and KE.

- **Chapter 6 - Algorithm Comparability**

  In this chapter, we present our approach to studying the comparability of software algorithms. We define a measure that assesses if two given algorithms are comparable or not. Experiments with a variety of clustering algorithms demonstrate the validity of our measure.

  In addition, we introduce a novel evaluation process for software clustering algorithms that utilizes multiple simulated authoritative decompositions. Experiments with a variety of software clustering algorithms are presented.

  Finally, we present a study that investigates differences between two of the more popular evaluation methods, MoJoFM and KE, in reduced search spaces.

- **Chapter 7 - Methods for selecting and improving software clustering algorithms**

  This chapter discusses the method we developed for selecting a suitable software clustering algorithm for a specific software system. In addition, it presents a method for improving software clustering algorithms. Finally, it describes a template for the formal description of software clustering algorithms.

- **Chapter 8 - Conclusions**

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

2 Background

Software clustering methodologies have been actively developed for more than twenty years. During this time, several software clustering algorithms have been published in the literature [MOTU93, CS90, AL97, LS97, SAP89, MV99, PHLR09]. Most of these algorithms have been applied to particular software systems with considerable success. Successful software clustering methodologies have significant practical value for software engineers.

A software engineer can apply a software clustering method for recovering lost information about a software system. Software clustering methods group entities of a software system, such as classes or source files, into subsystems, and compute views that will uncover buried facts about the software system. Such methods have been successfully applied to solve many software engineering problems [MNS95, ZKS04, Sch91].

There is 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 [Tze01]. Therefore, it is important to understand how good a solution created by a software clustering algorithm is. The research community has developed several methods for quality assessment of software clustering algorithms [TH99, ST04, MM01b, AL99, KE00].

In this chapter, we present an overview of the state of the art of software clustering methodology. We also outline directions for further research in software clustering, such as the development of better software clustering algorithms, or the improvement and evaluation of existing ones.

The structure of the chapter is as follows. The state of the art of software clustering algorithms is presented in Section 2.1. Evaluation of software clustering algorithms is presented in Section 2.3. Finally, Section 2.4 considers open issues related to software clustering methodology and concludes this chapter.

2.1 Software Clustering

In this section, we present the state of the art of software clustering research. We do so in the context of the more general framework of cluster analysis.

Cluster analysis is a group of multivariate techniques whose primary purpose is to group entities based on their attributes. Entities are classified according to predetermined selection criteria, so that similar objects are placed in the same cluster. The objective of any clustering algorithm is to sort entities into groups, so that the variation between clusters is maximized relative to variation within clusters.

The typical stages of cluster analysis techniques are as follows:

1. Fact Extraction (Section 2.1.1)

2. Filtering (Section 2.1.2)

3. Similarity Computation (Section 2.1.3)

4. Cluster Creation (Section 2.1.4)

5. Results Visualization (Section 2.1.5)

6. User Feedback Collection (Section 2.1.6)

The process typically repeats until satisfactory results are obtained.

We discuss each stage in detail in the following.

2.1.1 Fact Extraction

Before applying clustering to a software system, the set of entities to cluster needs to be identified. Entity selection depends on the objective of the method. For example, for program restructuring at a fine-grained level, function call statements are chosen as entities [XLZS04]. When the software clustering method applies to design recovery problems [MHH03, XT05, AT05, SMM02], the entities are often software modules. Classes [BT04] or routines [CCK00] can also be chosen as the entities.

After entities have been identified, the next phase is attribute selection. An attribute is usually a software artefact, such as a package, a file, a function, a line of code, a database query, a piece of documentation, or a test case. Attributes may also be high level concepts that encompass software artefacts, such as a design pattern. An entity may have many attributes. Selecting an appropriate set of attributes for a given clustering task is crucial for its success.

Most often, software artefacts are extracted directly from the source code, but sometimes artefacts are derived based on other kinds of information, such as binary modules,

software documentation etc. Most studies extract artefacts from various sources of input and store them in a language independent model [PDP+07]. These models can then be examined and manipulated to study the architecture of the software being built. For instance, FAMIX [DTD01] is used to reverse engineer object-oriented applications; its models include information about classes, methods, calls, and accesses. We often refer to these models as *factbases*.

The Tuple Attribute Language (TA) is a well-known format for recording, manipulating and diagramming information that describes the structure of large systems. The TA information includes nodes and edges in the graph, and attributes of these nodes and edges [Hol98]. It is also capable of representing typed graphs (graphs that can have more than one type of edge). Therefore, the TA language is not limited to recording only the structure of large systems. It can also express facts that are gathered from other sources.

The Dagstuhl Middle meta-model [TCLP04], GXL [HSSW06], MDG [Mit02], and RSF [MTW93] are other examples of factbase formats.

In the following, we present the most common inputs for the artefact extraction process.

Source Code

Source code is the most popular input for fact extraction. Many researchers [TGK99, BT04, Mit02] are using the source code as the only trusted foundation for uncovering lost information about a software system.

There are two conceptual approaches to extracting facts from source code: syntactic

and semantic. The syntactic (structure-based) approaches focus on the static relationships among entities. The exported facts include variable and class references, procedure calls, use of packages, association and inheritance relationships among classes etc.

Semantic approaches [KGMI04] include all aspects of a system's domain knowledge. The domain knowledge information present in the source code is extracted from comments, identifier names etc [KDG05].

Syntactic approaches can be applied to any software system, where as semantic approaches usually need to be customized to a specific software system due to domain specific assumptions. Knowledge spreading in the source code and absence of a robust semantic theory are other drawbacks of semantic approaches. However, the output from semantic approaches tends to be more meaningful than the one from syntactic approaches.

The software clustering community widely adopts structure-based approaches for large systems. In many cases the boundary between these approaches is not strict. Some clustering methodologies try to combine the strengths of both syntactic and semantic methods. The ACDC algorithm is one example of this mixed approach [Tze01].

Binary code

Some approaches work with the information available in binary modules. Depending on compilation and linkage parameters, the binary code may contain information, such as a symbol table that allows efficient fact extraction. This approach has three advantages:

1. It is language independent

2. Binary modules are the most accurate and reliable information (source code may have been lost or mismatched to a product version of binary modules. Source mismatch situations occur because of human mistakes, patches, intermediate/unreleased versions that are working in the production environment etc.)

3. Module dependency information is easy to extract from binary modules (Linkage information contains module dependency relations)

The main drawbacks of this approach are that binary meta-data information depends on building parameters, and that the implementation of the approach is compiler/hardware dependent. Also, binary code analysis cannot always discover all relationships. In particular, the Java compiler erases type parameter information and resolved references to final static fields of primitive types (constants) [DYM+08].

The binary code analysis method has been explored by the SWAG group [swaa]. SWAG has developed a fact extractor for Java called Javex. The output of the Java compiler is p-code. Javex is capable of extracting facts from the p-code and storing the facts using the TA format. Other researchers extracting information from bytecode are C. Lindig et al. [LS97] and J. Korn et al. [KCK99]. An interesting concept of collecting information about a software system is presented by Gang Huang et al. [HMY06]. Their approach collects facts from components. Components developed for frameworks such as CORBA/CCM, J2EE/EJB, COM+ etc. include interface information that can be easily extracted and explored.

Unfortunately, extraction tools based on source or binary code information cannot extract all facts about software systems. For instance, they cannot extract facts related to run-time characteristics of the software system. In addition, configuration information of component-based software systems, that are implemented and executed with the help of some common middleware (J2EE, COM+, ASP.NET etc.) is unavailable because it is stored in middleware configuration files that are not part of the source or binary code of the software system. This configuration information is important because it includes declaration of required resources, security realm and roles, component names for run-time binding and so on. Therefore, while the source and binary code contains important information about a software system, it does not contain complete information about the system.

Dynamic Information

Static information is often insufficient for recovering lost knowledge since it only provides limited insight into the runtime nature of the analyzed software; to understand behavioural system properties, dynamic information is more relevant [SS02]. Some information recovery approaches use dynamic information alone [WMFB+98, YGS+04], while others mix static and dynamic knowledge [SS02, PHLR09].

During the run-time of a software system, dynamic information is collected. The collected information may include:

1. Object construction and destruction

2. Exceptions/errors
3. Method entry and exit
4. Component interface invocation
5. Dynamic type information
6. Dynamic component names
7. Performance Counters and Statistics
   (a) Number of threads
   (b) Size of buffers
   (c) Number of Network Connections
   (d) CPU and Memory Usage
   (e) Number of Component Instances
   (f) Average, Maximum and Minimum Response Time

There are various ways of collecting dynamic information, such as instrumentation methods or third party tools (debuggers, performance monitors etc). Instrumentation techniques are based on introducing new pieces of code in many places to detect and log all collected events. Such techniques are language dependent, and not trivial to apply. The biggest concern with these techniques is ensuring that the newly generated software system has the same run-time behaviour as the original one. One option for implementing this approach is based on the Java ProbeKit. Probekit is a framework on the

Eclipse platform that you can use to write and use probes. Probes are Java code fragments that can be inserted into a program to provide information about the program as it runs. These probes can be used to collect run-time events needed for dynamic analysis. An alternative way to collect run-time information is to use debugger-based solutions [WMFB$^+$98, LN97, Tar00]. One advantage of using debuggers is that the source code remains untouched. Unfortunately, debugger information may not be sufficient to record the same aspects of the software system as instrumentation techniques. Also, some of the compilers cannot generate correct debug information when source code optimizations are enabled.

Today, managed runtime environments such as .NET and J2EE are popular paradigms. A virtual runtime environment allows gathering run-time information without modification of the source code. For instance, VTune analyzer [Dmi04] uses an industry standard interface in the JVM, the JVMPI (JVM Profiling Interface), for gathering Java-specific information. This interface communicates data regarding: memory locations and method names of JIT (just-in-time) emitted code, calls between Java methods, symbol information, etc.

Performance monitors allow the collection of statistical information about software systems such as CPU, memory usage, size of buffers etc. That information may uncover interesting behaviour relationships between software components.

Unfortunately, neither dynamic nor static information contain the whole picture of a software system. For instance, design information and historical information are not part of any input discussed so far.

Physical Organization

The physical organization of applications in terms of files, folders, packages etc. often represent valuable information for system understanding [LLG06]. Physical organization is not limited to the software development structure. It may also include the deployment structure and build structure. The deployment structure often follows industrial standards. Therefore, the location of a specific module provides valuable information about its responsibilities.

It is important to consider the physical organization of the software system because it often reflects the main ideas of the system design.

Human Organization

Human organization often reflects the structure of the system. Usually a developer is responsible for associated components. According to Conway: "Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations" [Con68].

Historical Information

Historical information explains the evolution of a software product. Recently, more research [Wuy01, CC05, FPG03] using historical information to reveal software design has appeared. Historical information is collected from version management systems, bug tracking systems, release notes, emails etc. Software evolution contains valuable infor-

mation for the solution of the software understanding problem [HH04]. For example, release notes contain a lot of valuable knowledge about product features and product releases.

Unfortunately, it is usually difficult to automatically/semi-automatically recover important system knowledge from historical sources due to the size and the lack of formatting of the extracted information.

Software Documentation

Software documents contain a lot of helpful information about software systems. However, they cannot be entirely trusted because of the reasons explained in Chapter 1. Facts extracted from software documents may not reflect the current state of the system. Therefore, the extracted facts should be validated with the current system. A.E. Hassan et al. [HH00] present such an approach. The idea of the method is to collect information about a software system from existing documentations and domain knowledge. The gathered information is then verified against the current stage of the software implementation.

Persistent Storage

Persistent repositories, such as databases, output files etc. contain information that can be helpful for software understanding. Developers often attempt to understand a software system by analyzing the application repositories. Software clustering methods should be able to utilize this information to their advantage as well [MJS+00].

Human Expertise

Humans may provide valuable facts based on their knowledge of requirement documents, high-level design and other sources.

Every input source has different advantages and disadvantages. Even though the end result of the clustering process will likely improve if the input factbase contains information from different sources [AATW07], the mixing of information from various sources is a challenging problem [XT05].

After the extraction process is finished, a filtering step may take place to ensure that irrelevant facts are removed, and the gathered facts are prepared for the clustering algorithm.

2.1.2 Filtering

The filter phase is the final stage of preparing a factbase. The main goal of this stage is to discard unnecessary information, calculate facts that are a composition of existing facts and apply a weighting scheme to the attributes.

For instance, all meaningless words extracted from source comments are discarded during the filter step [KDG05]. In an ideal situation, the factbase should be small and consist of enough information to ensure meaningful clustering.

The information contained in the final factbase may be dependent on assumptions of a specific software clustering algorithm. Some algorithms expect that the factbase includes only relations between modules [SMM02]; other algorithms do not make any assump-

tions about the facts [AT05].

The software research community most often applies the following filters:

1. Utilities

2. Granularity projection

These filters process the collected factbase and discard unnecessary information. The study presented in [LZN04] shows that using a different methodology for the construction of the final factbase may affect results significantly.

This discovery emphasizes the importance of preparing the final factbase. Several tools have been developed that allow fact manipulation. For instance, Grok [Hol98] is specifically developed for the manipulation of facts extracted from a software system. Other methods utilize SQL or Prolog for fact manipulations.

Utilities

In many cases, modules containing utilities do not follow common design practices such as high cohesion and low coupling. For example, utility modules may include drivers, commonly used methods etc. As a result, utilities may require special treatment. Some software clustering algorithms are not affected by utilities [AT05]; others, such as Bunch, may be affected [MJ06]. Some authors have argued that removing utilities improves the overall results [MOTU93]. Others suggest considering utilities as a class that should be kept for further investigation, because it plays an important role in the implementation of the overall solution by allowing communication between other classes [MJ06]. Therefore,

the research community has not reached a conclusion about the best approach to dealing with utilities.

In cases where the software clustering method is affected by utilities, a utility filter should be applied. Such a filter identifies utilities based on the facts present in the factbase. There are different ways to do this. Adelwahab Hamou-Lhadj et al. [HLBAL05] identified utilities as classes that have many direct client classes. Wen and Tzerpos [WT05] present a utility module detection method where a module is identified as a utility if it is connected to a large number of sub-systems (clusters) rather than entities.

Granularity projection

A large number of software clustering algorithms utilize mostly the relations between modules/classes. The goal of a granularity projection filter is to use low level facts, such as function calls or variable references, in order to calculate dependencies between classes, and then remove the low level facts from the factbase.

After the final factbase is constructed, the next step is to compute similarities.

2.1.3 Similarity computation

Most software clustering methods initially transform a factbase to a data table, where each row describes one entity to be clustered. Each column contains the value for a specific attribute. Table 2.1 presents an example of a data table. It contains information about file-to-file relationships, where each entity and each attribute are files of the software system. The values of the attributes are calculated based on the dependencies between the files. In

this example, file $f1.c$ depends on file $f3.c$ (possibly by calling a function in $f3.c$), while file $f3.c$ does not depend on $f1.c$.

	$f1.c$	$f2.c$	$f3.c$
$f1.c$	1	1	1
$f2.c$	0	1	0
$f3.c$	0	0	1

Table 2.1: Data Table Example.

In most cases, a different column corresponds to a different attribute. Sometimes, different columns contain information about related attributes. For example, categorical attributes[1] are often represented as a composition of multiple binary attributes [AT05]. Table 2.2 is an example of such a situation. It is an extension of Table 2.1 with a new categorical attribute representing developer names.

	$f1.c$	$f2.c$	$f3.c$	Alice	Bob
$f1.c$	1	1	1	1	0
$f2.c$	0	1	0	0	1
$f3.c$	0	0	1	1	0

Table 2.2: Categorical Attribute Presented as Binary Data.

Categorical data can be represented in compact form as well. Table 2.3 is an example

[1] A categorical attribute is an attribute with a finite number of values (in practice, a *small* number of *discrete* values, such as developer names) and no inherent ranking.

of compact form representation of the same data as in Table 2.2.

	$f1.c$	$f2.c$	$f3.c$	Developer
$f1.c$	1	1	1	Alice
$f2.c$	0	1	0	Bob
$f3.c$	0	0	1	Alice

Table 2.3: Categorical Attribute in compact form.

Similarity function

Clustering algorithms are based on a similarity function between entities [JD88]. However, some algorithms, such as hierarchical agglomerative ones are applying the similarity function explicitly, while others, such as search-based algorithms are using the similarity function only implicitly.

The most common type of similarity functions are resemblance coefficients. Other similarity functions include probabilistic measures and software specific similarities.

Resemblance coefficients

A resemblance coefficient can be binary, numerical, categorical or mixed. Binary and categorical resemblance coefficients are called qualitative, while numerical ones are called quantitative. The selection of the appropriate approach for the calculation of the resemblance coefficients depends on the type of input data matrix. A qualitative resemblance

coefficient is calculated based on qualitative input data (binary or categorical); a quantitative resemblance coefficient is based on quantitative input data.

Binary Resemblance. The intuition behind the calculation of resemblance coefficients is to measure the amount of relevant matches between two entities. In other words, the more relevant matches there are between two entities, the more similar the two entities are. There are different methods for counting relevant matches and many formulas exist to calculate resemblance coefficients [Rom90]. Some well-known examples are given in Table 2.4. In these formulas, a represents the number of attributes that are "1" in both entities, b and c represent the number of attributes that are "1" in one entity and "0" in the other, and d represents the number of attributes that are "0" in both entities.

Similarity measure	Formula
Simple matching coefficient	$\frac{a+d}{a+b+c+d}$
Jaccard coefficient	$\frac{a}{a+b+c}$
Sorenson coefficient	$\frac{2a}{2a+b+c}$
Rogers and Tanimoto	$\frac{a+d}{a+2(b+c)+d}$
Russel and Rao	$\frac{a}{a+b+c+d}$

Table 2.4: Examples of binary resemblance coefficient.

A binary resemblance coefficient that is suitable for software clustering will ideally include the following two properties:

1. 0-0 matches are ignored, i.e. d is not part of the formula. The joint lack of attributes

between two entities should not be counted toward their similarity [LZN04]

2. Heavier weight is assigned to more important factors[Dha95]

A binary resemblance coefficient that fits these software clustering assumptions is the Sorenson coefficient [LZN04]. Research [DB00] concludes that Jaccard and Sorenson have performed well, but the authors recommend using the Jaccard algorithm because it is more intuitive.

Categorical Resemblance. There are similarities between binary and categorical resemblance coefficients. The calculation of a categorical resemblance coefficient, similar to that of a binary resemblance coefficient, is based on the number of matches between two entities. When categorical attributes are represented as a set of binary attributes, then the calculation of the categorical coefficient is based on the calculation of the binary resemblance coefficient. When categorical attributes are represented in compact form, then the categorical coefficient is calculated based on the simple matching formula (See Table 2.4).

Quantitative Resemblance. Quantitative resemblance coefficients calculate distance between entities. Each entity is represented as a vector. For instance, entity $f1.c$ from Table 2.1 can be represented as vector $(1, 1, 1)$. Its distance to other vectors can be calculated using formulas such as:

1. Euclidean: $\sqrt{\sum_{i=1}^{n}((x_i - y_i)^2)}$

2. Maximum: $max|x_i - y_i|$

3. Manhattan: $\sum_{i=1}^{n}(|x_i - y_i|)$

Mixed Resemblance. An entity in the data table may be described by more than one type of attributes. At the same time, some values in the data table may be missing. For those cases, the widely used general similarity coefficient was developed by Gower [Gow71]. Let x and y denote two entities and described over d attributes. Then, the general similarity coefficient $S_{Gower}(x, y)$ is defined as:

$$S_{Gower}(x, y) = \frac{1}{\sum_{k=1}^{d}(w(x_k, y_k)} \sum_{k=1}^{d}(w(x_k, y_k)s(x_k, y_k))$$

where $s(x_k, y_k)$ is a similarity component for the kth attribute and $w(x_k, y_k)$ is either one or zero, depending on whether or not a comparison is valid for the kth attribute of the entities.

Probabilistic Measures

Probabilistic measures are based on the idea that agreement on rare matches contributes more to the similarity between two entities than agreement on more frequent ones [Wig97]. The probabilistic coefficients require the distribution of the frequencies of the attributes present over the set of entities. When this distribution is known, a measure of information or entropy can be computed for each attribute. Entropy is a measure of disorder; the smaller the increase in entropy when two (sets of) entities are combined, the more similar the two entities are. For a more detailed discussion on probabilistic coefficients, we refer to [SS73].

Software Specific Similarity

There are also similarity functions that have been developed specifically for the software clustering problem. Schwanke et al. [Sch91] introduced the notion of using design principles, such as low coupling and high cohesion. Koschke [Kos00] has developed an extension of Schwanke's metric-based hierarchical clustering technique. The Koschke similarity functions include global declarations, function calls etc. Also, the similarity method is considering name similarities between identifiers and filenames. Choi and Scacchi [CS90] also describe a similarity function based on maximizing the cohesiveness of clusters.

2.1.4 Cluster Creation

At this point all preparation steps are completed, and the clustering algorithm can start to execute. In this section, we discuss various software clustering algorithms. Wiggerts [Wig97] suggests the following classification of software clustering algorithms:

1. Graph-Theoretical Algorithms
2. Construction Algorithms
3. Optimization Algorithms
4. Hierarchical Algorithms

Graph-Theoretical Algorithms

This class of algorithms is based on graph properties. The nodes of such graphs represent entities and the edges represent relations. The idea of graph algorithms is to find

sub-graphs which will form the clusters. Special kinds of sub-graphs like connected components, cliques and spanning trees are used to derive clusters. The two most common types of graph-theoretical clustering algorithms are aggregation algorithms and minimal spanning tree algorithms.

Aggregation algorithms reduce the number of nodes (representing entities) in a graph by merging them into aggregate nodes. The aggregates can be used as clusters or can be the input for a new iteration resulting in higher level aggregates.

Common graph reduction techniques are (i) the notion of the neighbourhood of a node [vL93] , (ii) strongly connected components[BS91] and (iii) bi-components [BS91].

Minimal spanning tree algorithms (MST) algorithms begin by finding an MST of the given graph. Next, they either interactively join the two closest nodes into a cluster or split the graph into clusters by removing "long" edges. The classic MST algorithm is not suited for software clustering due to the fact that the algorithm tends to create a few large clusters that contain many entities while several other entities remain separate [Tri01]. Markus Bauer et al. suggest a two-pass modified MST algorithm [BT04]. The first pass, which follows the classic MST concept, iteratively joins the two closest nodes into a cluster while the second pass assigns the remaining un-clustered entities to the cluster they are the "closest" to.

Construction Algorithms

The algorithms in this category assign the entities to clusters in one pass. The clusters may be predefined (supervised) or constructed as part of the assignment process (unsupervised). Examples of construction algorithms include the so-called geographic techniques and the density search techniques. A well known geographic technique is the bisection algorithm, which at each step divides the plain in two and assigns each entity according to the side that it lies on.

An algorithm based on fuzzy sets was presented in [GL70]. An ordering is defined on entities determined by their grade of membership (defined by the characteristic function of the fuzzy set). Following this order, each entity is either assigned to the last initiated cluster or it is used to initiate a new cluster, depending on the distance to the entity which was used to initiate the last initiated cluster.

Mode analysis [Wis69] is another example of a construction clustering algorithm. For each entity, it computes the number of neighbouring entities that are "closer" than a given radius. If this number is large enough, then the algorithm clusters the entities together.

Optimization algorithms

An optimization or improvement algorithm takes an initial solution and tries to improve this solution by iterative adaptations according to some heuristic. The optimization method has been used to produce both hierarchical [Lut01] and non-hierarchical [MM09] clustering.

A typical non-hierarchical clustering optimization method starts with an initial partition derived based on some heuristic. Then, entities are moved to other clusters in order to improve the partition according to some criterion. This relocating goes on until no further improvement of this criterion takes place. Examples of clustering optimization methods are presented in [CDH+03].

One of the famous representatives of the optimization class of algorithms is ISODATA [And73]. Its effectiveness is based on the successful initial choice of values for seven parameters that control factors such as the number of expected clusters, the minimum number of objects in the cluster, the maximum number of iterations 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.

Genetic Techniques

Genetic clustering algorithms are optimization algorithms that perform an optimization that follows the concept of Genetic Algorithms (GA). Genetic algorithms are randomized search and optimization techniques guided by the principles of evolution and natural genetics, having a large amount of implicit parallelism. Genetic algorithms are characterized by attributes such as the objective (optimization) function, the encoding of the input data, the genetic operators (reproduction rules such as crossover and mutation), and population size.

A typical genetic algorithm runs as follows:

1. Select a random population of partitions

2. Generate a new population by selecting the best individuals according to the objective function and reproducing new ones by using the genetic operations

3. Repeat step 2 until a chosen stop criterion is satisfied

D. Doval et al. [DMM99] present a schema for mapping the software clustering problem to a genetic problem. The quality of a partition is determined by calculating a modularization quality function. There are several ways for the calculation of modularization quality. In general, the modularization quality measures the cohesion of clusters and their coupling. The result of the algorithm is a set of clusters for which an optimal modularization quality was detected; in other words, clusters that feature an optimal tradeoff between coupling and cohesion. The D. Doval et al. genetic algorithm has been implemented as part of the Bunch software clustering tool [MM06].

Ali Shokufandeh et al. claim that the GA Bunch algorithm is especially good at finding a solution quickly, but they found that the quality of the results produced by Bunch's hill-climbing algorithms is typically better [SMDM05].

Olaf Seng et al. proposed an improved software clustering GA algorithm [SBBP05] based on Falkenauer's class of GA algorithms [Fal98]. According to the author, the algorithm is more stable than Bunch and its objective function evaluates additional quality properties of the system's decomposition, such as individual subsystem size.

Hill-Climbing Search

Hill-Climbing clustering algorithms perform the following steps:

1. Generate a random solution
2. Explore the neighbourhood of the solution attempting to find a neighbour better than the solution. Once a neighbour is found, it becomes the solution.
3. Repeat step 2 until there is no better neighbour

Hill-climbing search methods have been successfully employed in various software clustering algorithms [CDH+03]. Mitchell's PhD dissertation [Mit02] shows promising results in terms of the quality and performance of hill-climbing search methods. His approach has been implemented as part of the Bunch software clustering tool [MM02, MMR+98].

Bunch starts by generating a random partition of the module dependency graph. Then, entities from the partition are regrouped systematically by examining neighbouring partitions in order to find a better partition. When an improved partition is found, the process repeats, i.e. the found partition is used as the basis for finding the next improved partition. The algorithm stops when it cannot find a better partition. The objective function is the modularization quality function used also in Bunch's genetic algorithm. Kairash Mahdavi et al. present an improvement to existing hill-climbing search approaches based on applying a hill-climbing algorithm multiple times. The proposed approach is called multiple hill climbing [MHH03]. In this approach, an initial set of hill-climbing searches is performed. The created partitions are used to identify the common features of each solution. These common features form building blocks for a subsequent

hill climb. The authors found that the multiple hill-climbing approach does indeed guide the search to higher peaks in subsequent executions.

Spectral Clustering Technique

A typical spectral clustering algorithm follows these steps:

1. Build the Laplacian matrix corresponding to the system's dependency graph
2. Determine the dominant eigenvalues and eigenvectors of the Laplacian matrix
3. Use these to compute the clustering

Some researchers have adapted spectral graph partitioning to the decomposition of software systems [SMDM05, Xan06]. This is based on a construction graph that represents relations between entities in the explored system. Spectral clustering algorithms are recursive. Each iteration splits the graph to two sub-graphs and calculates the new value of the objective function. The recursion terminates as soon as the objective function stops improving.

First, the Laplacian matrix is calculated and the smallest non-zero eigen-value is found. This value will be used for the calculation of the characteristic vector[2], which is used to partition the graph. The graph is divided into sub-graphs based on the values of the characteristic vector. The algorithm uses the entries of the characteristic vector to split the entities so that the break-point maximizes the goal function. If the bisection improves

[2]The characteristic vector is a n-dimensional vector $(x_1, ..., x_n)$ that defines two clusters. Entities whose x_i is 0 belong in the first cluster, entities whose x_i is 1 belong in the second cluster.

the objective function, then the algorithm goes to the next iteration by splitting each subgraph obtained in the previous step recursively. If splitting does not improve the solution, then the algorithm stops.

Ali Shokoufandeh et al. [Xan06] developed a spectral software clustering method that guarantees that the constructed partition is within a known factor of the optimal solution. The objective function used is the same as in the Bunch algorithms.

Clumping Techniques

Another form of algorithms that perform optimization are the so-called clumping techniques [Wig97]. In each iteration, one cluster is identified. Repeated iterations discover different clusters (or clumps) which may overlap. A negative aspect of this method is that finding the same clump several times cannot be avoided completely.

Hierarchical algorithms

There are two categories of hierarchical algorithms: agglomerative (bottom-up) and divisive (top-down).

Divisive algorithms start with one cluster that contains all entities and divide the cluster into a number (usually two) of separate clusters at each successive step. Agglomerative algorithms start at the bottom of the hierarchy by iteratively grouping similar entities into clusters. At each step, the two clusters that are most similar to each other are merged and the number of clusters is reduced by one.

According to [KR90], divisive algorithms offer an advantage over agglomerative algorithms because most users are interested in the main structure of the data which consists of a few large clusters found in the first steps of divisive algorithms. Agglomerative algorithms start with individual entities and work their way up to large clusters which may be affected by unfortunate decisions in the first steps. Agglomerative hierarchical algorithms are most widely used however. This is because it is infeasible to consider all possible divisions of the first large clusters [Wig97].

Agglomerative Algorithms

Agglomerative algorithms perform the following steps [MB04]:

1. Compute a similarity matrix
2. Find the two most similar clusters and join them
3. Calculate the similarity between the joined clusters and others obtaining a reduced matrix
4. Repeat from step 2 until two clusters are left

The above process implies that there is a way to calculate the similarity between an already formed cluster and other clusters/entities. This is done via what is called the update rule function. Suppose that cluster i and cluster j are joined to form cluster ij. Typical update rule functions are:

1. Single linkage: $sim(ij, k) = min(sim(i, k), sim(j, k))$

2. Complete linkage: $sim(ij, k) = max(sim(i, k), sim(j, k))$

3. Average linkage: $sim(ij, k) = \frac{1}{2}[sim(i, k) + sim(j, k)]$

O. Maqbool et al. [MB04] has concluded that for software clustering, the complete linkage update rule gives the most cohesive clusters. The same work introduced a new update rule called weighted combined linkage that provided better results than complete linkage. This result was achieved by applying the unbiased Ellenberg measure [MB04] and utilizing information regarding the number of entities in a cluster that access an artefact, thereby substantially reducing the number of arbitrary decisions made during the algorithm's clustering process.

UPGMA (Unweighted Pair Group Method with Arithmetic mean) [Rom90] is an agglomerative hierarchical method used in bioinformatics for the creation of phylogenetic trees. Chung-Horng Lung et al. [LZN04] have shown applications of the UPGMA method in the software clustering context.

Andritsos et al. [AT05] presented the Scalable Information Bottleneck (LIMBO) algorithm, an agglomerative hierarchical algorithm that employs the Agglomerative Information Bottleneck algorithm (AIB) for clustering. LIMBO uses an information loss measure to calculate similarity between entities. At every step, the pair of entities that would result in the least information loss is chosen.

ACDC [TH00a] is a hierarchical clustering algorithm that does not follow a standard schema. It cannot be assigned to the agglomerative or divisive category because the algorithm does not have an explicit iterative split or merge stage. ACDC uses patterns

that have been shown to have good program comprehension properties to determine the system decomposition. ACDC systematically applies these subsystem patterns to the software structure. This results into most of the modules being placed into hierarchical categories (subsystems). Then, ACDC uses an orphan adoption algorithm [TH97] to assign the remaining modules to the appropriate subsystem.

We have discussed various algorithms and similarity techniques that have been adapted to the software clustering context.

An important observation is that there are two different conceptual approaches to developing a software clustering methodology. The first one attempts to develop a sophisticated structure discovery approach such as ACDC, Bunch etc. The second approach concentrates more on developing similarity functions [Kos00, Sch91]. An important open research question is: Which approach is the most promising for the future of software clustering?

2.1.5 Results Visualization

The output of the cluster discovery stage needs to be presented in a user friendly manner. The challenge is to present a large amount of data with its relations in such a way that a user can understand and easily work with the data. The presentation of software clustering results is an open research question that is related to scientific visualization (the computer modeling of raw data) and Human-Computer Interaction.

The software reverse engineering community has developed various tools for the presentation of the output of software clustering. These often include the traditional soft-

ware representation of a graph. Such tools include CodeCrawler[LD03] , Rigi[MTW93] and LSEdit[swaa]. Rigi and LSEdit allow manual modification of the clustering result.

A different research direction is to develop visualizations of software systems by using metaphors and associations. For instance, Code City [WL07] is a 3D visualization tool that uses the City metaphor to visualize software systems. It is an integrated environment for software analysis, in which software systems are visualized as interactive, navigable 3D cities. The classes are represented as buildings in the city, while the packages are depicted as the districts in which the buildings reside. The visible properties of the city artefacts depict a set of chosen software metrics.

A reverse engineer will understand the output produced by a software clustering method better if the visualization techniques improve. Once a user has understood the clustering results, they may be able to provide feedback that can be used to refine method parameters. Therefore, effective visualization of results opens the way for the development of interactive software clustering algorithms where a user can iteratively assess the current results and steer the software clustering process. This concept is described in the next section.

2.1.6 Feedback

During the clustering process, an expert user should be able to instruct the tool on how to improve the overall solution. Clustering algorithms that are able to process user feedback are called semi-automatic clustering algorithms. Christl et al. [CKS07] present a semi-automatic algorithm that allows the mapping of hypothesized high-level entities to

source code entities. Rainer Koschke [Kos00] created a semi-automatic clustering framework based on modified versions of the fully automatic techniques he investigated. The goal of Koschke's framework is to enable a collaborative session between his clustering framework and the user. The clustering algorithm does the processing, and the user validates the results.

Semi-automatic cluster analysis algorithms are more complicated than the fully automatic ones. They produce results that are closer to the expectations of the user than those of fully automatic methods. This is both an advantage and a disadvantage. A software engineer may explore different aspects of the software system by validating clustering results and providing feedback to the semi-clustering algorithm. On the other hand, the result of semi-automatic clustering may not reflect the actual state of the software system, because in many cases the software engineer may have wrong or incomplete understanding of the current state of the system. This drawback may explain the fact that the reverse engineering research community is mostly developing automatic clustering algorithms.

2.2 Definitions

We pause momentarily to define terms that we will use frequently throughout the thesis. The output of a software clustering algorithm is either a flat or a nested decomposition of a software system.

A *flat decomposition* does not contain nested clusters. There is one level of clusters and one level of entities. An example of a flat decomposition is shown in Figure 2.1.

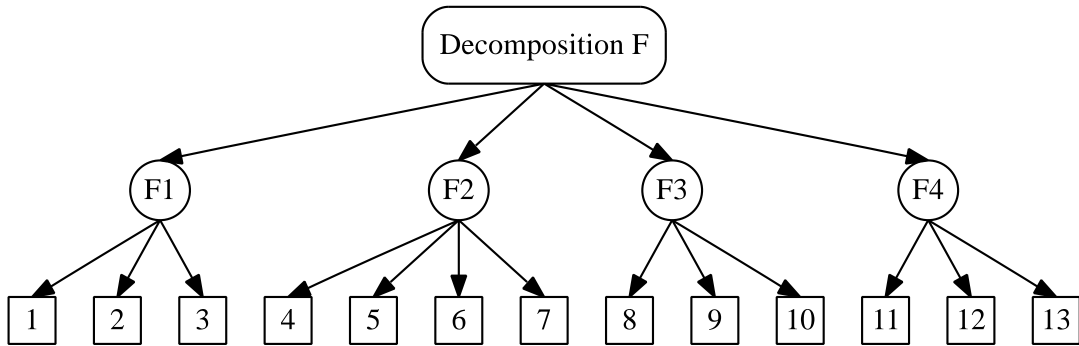


Figure 2.1: An example of a flat decomposition.

A *nested decomposition* is a decomposition that may contain nested clusters. An example of a nested decomposition is shown in Figure 2.2.

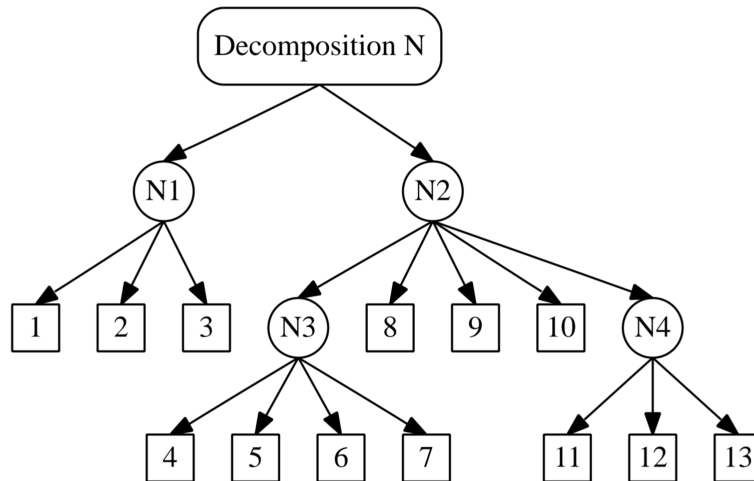


Figure 2.2: An example of a nested decomposition.

Moreover, we distinguish two types of software decompositions depending on who created them. Decompositions that are constructed by software clustering algorithms are called *automatic decompositions*. Decompositions that are constructed by an expert in the software system are called *authoritative decompositions*.

Different experts may construct distinct software decompositions, since a software system may have several equally valid decompositions. In the research community, an authoritative decomposition is considered as one of many possible decompositions that can help software maintainers understand a software system. As a result, clustering researchers should experiment with all available authoritative decompositions. Experimental results typically attempt to determine the ability of a clustering algorithm to produce decompositions that are similar to an authoritative one. Since a software clustering algorithm can only create particular types of decompositions it is important to remember that experimental results are applicable only in the context of a particular authoritative decomposition. In this thesis, all experimental results calculated against an authoritative decomposition are implied to be valid only with regard to this authoritative decomposition.

2.3 Evaluation of Clustering Algorithms

Software clustering researchers have developed several evaluation methods for software clustering algorithms. This research is important because

1. Most software clustering work is evaluated based on case studies. It is important that the evaluation technique is not subjective.

2. Evaluation helps discover the strengths and weaknesses of the various software clustering algorithms. This allows the development of better algorithms through addressing the discovered weaknesses.

3. Evaluation can help indicate the types of system that are suitable for a particular algorithm. For instance, Mitchell et al. [MM01b] think that Bunch may not be suitable for event-driven systems.

The importance of evaluating software clustering algorithms was first stated in 1995 by Lakhotia and Gravely [LG95]. Since then, many approaches to this problem have been published in the literature. These can be divided in two categories:

1. Based on an authoritative decomposition

2. Not based on an authoritative decomposition

2.3.1 Evaluation based on authoritative decomposition

The main principle of evaluation based on an authoritative decomposition is that a clustering produced by an algorithm should resemble the clustering produced by some authority. Therefore, such evaluation methods provide the means to calculate the quality of an automatic decomposition by comparing it to the authoritative one.

The evaluation methods can be divided in two categories. The first category evaluates a software clustering approach based on the comparison between the authoritative decomposition and the automatic decomposition. A typical example of such a method is the MoJo distance [TH99]. Most of the evaluation methods that have been developed belong to this category.

Evaluation methods of the second category focus on the evaluation of a specific stage of the software clustering process. Such a method calculates the quality of a specific stage

based on analysis of its inputs and outputs. For instance, an evaluation method, that focuses on the evaluation of the analysis phase, will take into account the input meta-model, compare the authoritative decomposition and the produced software clustering decomposition, and then calculate a number that reflects the quality of the produced decomposition. A typical example of such a method is EdgeSim [MM01a].

The main reason for the development of both types of evaluation methods is that the quality of a produced decomposition depends on the:

1. Selection of an appropriated clustering algorithm – A different clustering algorithm produces different outputs from the same factbase.

2. Selection of input parameters – A clustering algorithm might produce different outputs depending on selected input parameters, such as similarity function, input meta-model etc.

An orthogonal categorization of software clustering evaluation methods divides them into two classes:

1. Evaluation of software clustering algorithms that produce flat decompositions

2. Evaluation of software clustering algorithms that produce nested decompositions

The evaluation of algorithms that produce flat decompositions is better studied than the evaluation of algorithms that produce nested decompositions.

MoJo Family

Tzerpos and Holt developed a distance measure called MoJo [TH99]. It attempts to capture the distance between two decompositions as the minimum number of Move and Join operations one has to perform in order to transform one into the other. A Move operation involves relocating a single entity from one cluster to another (or to a new cluster), while a Join operation takes two clusters and merges them into a single cluster. MoJo distance is non-symmetric.

Wen et al. [WT03] presented an algorithm that calculates MoJo distance in polynomial time. They also introduced the MoJoFM effectiveness measure which is based on MoJo distance but produces a number in the range 0-100 which is independent of the size of the decompositions [WT04b].

Precision/Recall

Precision and Recall are standard metrics in Information Retrieval. They have been applied to the evaluation of software clustering by Anquetil et al. [AL99]. The method calculates similarity based on measuring intra pairs, which are pairs of entities that belong to the same cluster. The authors suggest to calculate precision and recall for a given partition as follows:

- Precision – Percentage of intra pairs proposed by the clustering method, which are also intra pairs in the authoritative decomposition
- Recall – Percentage of intra pairs in the authoritative decomposition, which are also

intra pairs in the decomposition proposed by the clustering method

Mitchell et al. [MM01b] explain a drawback of the Precision/Recall metrics: "An undesirable aspect of Precision/Recall is that the value of this measurement is sensitive to the size and number of clusters".

Koschke and Eisenbarth

Koschke and Eisenbarth [KE00] have presented a way of quantitatively comparing automatic and authoritative partitions that establishes corresponding clusters (Good match), handles partially matching clusters (OK match), tolerates smaller divergences, and determines an overall recall rate. Their approach is as follows

1. Identify immediately corresponding clusters (Good match)
2. Identify corresponding sub-clusters, i.e. where part of a cluster of one partition corresponds to part of a cluster in the other partition (OK match)
3. Measure accuracy of the correspondences between two partitions

This metric has two drawbacks:

1. It does not penalize the software clustering algorithm when an automatic decomposition is more detailed than the authoritative one
2. The recall rate does not distinguish between a Good match and an OK match.

The authors have also developed benchmark scenarios for the evaluation of software clustering and the calibration of software clustering parameters. For the remainder of the thesis, we refer to their quality metric as KE.

EdgeSim and MeCl

Mitchell and Mancoridis [MM01a] developed the first method for the evaluation of a software clustering algorithm in conjunction with an input meta-model. They introduced a similarity (EdgeSim) and a distance (MeCl) measurement that consider relations between entities as an important factor for the comparison of software system decompositions. The EdgeSim similarity measurement normalizes the number of intra- and inter-cluster edges that agree between two partitions. The MeCl similarity measurement determines the distance between a pair of partitions by first calculating the "clumps", which are the largest subset of modules from both partitions that agree with respect to their placement into clusters. Once the clumps are determined, a series of Merge operations are performed to convert the first partition into the second one. The actual MeCl distance is determined by normalizing the number of Merge operations.

Unfortunately, these two measures can be applied only when the input meta-model is a dependency graph. Another drawback of the method is the assumption that the authoritative decomposition is a direct reflection of the dependencies in the factbase. It is difficult to confirm that this assumption holds because the system expert that has constructed the authoritative decomposition has certainly used additional knowledge that may contradict this assumption. Wen et al. [WT04a] explain the main drawback of Ed-

geSim and MeCl: "it only considers the misplacement of edges without considering the misplacement of the objects in the wrong clusters".

In the next section, we present methods that allow the evaluation of software clustering algorithms in the absence of an authoritative decomposition.

2.3.2 Evaluation methods not based on an authoritative decomposition

The main drawback of methods that require an authoritative decomposition is that they assume that such a decomposition exists. To construct such a decomposition for a middle-size software system is a challenging task. Various research studies [MM01a, KE00, GK00] deal with the construction of an authoritative decomposition. This work addresses two questions:

1. What is the right process for the construction of an authoritative decomposition?

2. Why is the constructed decomposition authoritative?

Arun Lakhotia [LG95] said, "If we know the subsystem classification of a software system, then we do not need to recover it". Another aspect of the problem is that after a researcher constructs an authoritative decomposition, they may find out that the authoritative decomposition cannot be used for evaluation purposes. For instance, if the authoritative decomposition is flat then it is not suitable to evaluate algorithms that produce nested decompositions. Finally, a software system may have a number of different authoritative decompositions.

To overcome these problems several evaluation methods that evaluate a software clustering approach based solely on its output have been developed [AL99, WHH05]. Jingwei et al. [WHH05] suggest comparing software clustering algorithms based on two criteria:

1. Stability
2. Extremity of Cluster Distribution

Stability

Stability reflects how sensitive is a clustering approach to perturbations of the input data. Vijay V. Raghavan [Rag82] has shown the importance of developing such a metric. The intuition behind stability in software clustering is that similar clustering decompositions should be produced for similar versions of a software system. A stability function measures the percentage of changes between produced decompositions of successive versions of an evolving software system. Under conditions of small changes between consecutive versions, an algorithm should produce similar clustering [WHH05].

Tzerpos et al. [TH00b] defines a stability measure based on the ratio of the number of "good" decompositions to the total number of decompositions produced by a clustering algorithm. A decomposition, which is obtained from a slightly modified software system, is defined as "good" iff the MoJo distance between the decomposition and decomposition obtained from the original software system is not greater than 1% of the total number of entities.

Jingwei Wu et al. [WHH05] argued that a drawback of the aforementioned stability

measure is that 1% seems too optimistic in reality. In the same paper, the authors suggest a relative stability measure. This measure allows one to say that one algorithm is more stable than another with regard to a software system. To calculate the relative score of two clustering algorithms, the authors suggest constructing sequences of MoJo values calculated based on comparing two consecutive members of the sequence of decompositions obtained from a software system. Then, based on the two sequences of MoJo values, a number is calculated that represents a relative score of one algorithm over the other.

Extremity of Cluster Distribution

An interesting property of a clustering algorithm is the size of the clusters it produces. The cluster size distribution of a clustering algorithm should not exhibit extremity. In other words, a clustering algorithm should avoid the following situations:

1. The majority of files are grouped into one or few huge clusters

2. The majority of clusters are singletons

Jingwei Wu et al. [WHH05] presented a measure called NED (non-extreme distribution) that allows to evaluate the extremity of a cluster distribution. NED is defined as the ratio of the number of entities contained in non-extreme clusters to the total number of the entities.

Nicolas Anquetil et al. [AL99] have investigated the causes of extreme cluster distribution. They found that poor input meta-model and unsuitable similarity metrics can cause such results. For instance, the majority of the clusters being singletons is a sign of a bad

descriptive attribute. A reason that the majority of files are grouped into one or few huge clusters is a consequence of an ill-adapted algorithm or similarity metric.

The main drawback of both presented methods is that they can only identify "bad" software decompositions. It is easy to develop an algorithm that produces decompositions that are stable and non-extreme, while being entirely useless for program understanding purposes.

Mitchell and Mancoridis [MM01b] developed a framework called CRAFT for the evaluation of software clustering algorithms without authoritative decomposition. The proposed evaluation process consists of two phases. The first phase is the construction of an authoritative decomposition. The CRAFT framework automatically creates an authoritative decomposition based on common patterns produced by various clustering algorithms. The initial step of the process is to cluster the target system many times using different clustering algorithms. For each clustering run, all the modules that appear in the same cluster are recorded. Using this information, CRAFT exposes common patterns in the results produced by different clustering algorithms and it constructs an authoritative decomposition. The assumption of the approach is that agreement across a collection of clustering algorithms should reflect the underlying structure of the software system.

The second phase is to compare the created authoritative decomposition with the automatic decomposition by applying an evaluation method that requires an authoritative decomposition. The advantage of CRAFT is that it is applicable when an authoritative decomposition does not exist. The main drawback of the method is that the automatically produced authoritative decomposition may be unacceptable according to a software ex-

pert. It can only be as good as the algorithms it utilized.

In this section, we presented an overview of the important methods for the evaluation of software clustering algorithms. Most methods require an authoritative decomposition (either flat or nested). Methods that do not require an authoritative decomposition can be used to weed out bad clustering results, i.e. to select appropriate parameters for the clustering algorithms.

2.4 Conclusions

We have presented the state of the art of software clustering methods. We discussed different algorithms and various approaches of gathering information utilizing different meta-models.

In this section, we present open research challenges in software clustering. We have selected research questions that are related to the following topics:

1. Attribute Gathering
2. Cluster Discovery
3. Visualization of Results
4. User Feedback
5. Evaluation of Clustering Algorithms

Before we present each topic in detail, we bring attention to the current situation with software clustering tools. The research community has shown many advantages to us-

ing software clustering methods in different software engineering areas. Unfortunately, software clustering methodologies are not widely accepted in the industrial environment. There is still a long way to go before software clustering methods become an effective and integral part of the IDE [MJS+00]. Currently, the existing toolset is not mature enough to be used as part of a developer's daily activity.

Fact Extraction

We discussed various input sources for the attribute gathering process. Typically, a clustering method will utilize only one of these sources, such as source or binary code. However, it is possible to construct input meta-models that combine information from different sources. The reason this does not happen more often in practice is the lack of an established methodology. Such a methodology has to answer various questions including:

1. Is it important to mix various sources? How important is a good meta-model?
2. Which source(s) contain(s) the most important information?
3. What is an effective way to represent artefacts gathered from various input sources in the same format in one meta-model? In other words, we have to define a language that allows the expression of an artefact extracted from any input source.
4. What is a reasonable weight scheme for artefacts combined from various sources? Does it vary based on factors such as the type of system or its development methodology?

5. What are other input sources for extraction of artefacts that could be integrated into the input meta-model?

These are the key questions for the attribute gathering stage. The answers to those questions would allow the development of a good meta-model, which should result in better clustering results. In addition, these answers would allow the research community to merge their efforts to develop a common methodology for gathering attributes. Today, efforts are split on developing advanced methods for extracting data from specific input sources.

Cluster Discovery

The cluster discovery process encompasses the similarity computing and cluster creation phases. It is a complicated process that has been discussed in many research studies. Still, several key questions remain unanswered. We present a list of as yet unsolved problems that pose interesting challenges to researchers in the area:

1. The best direction towards improving this process needs to be determined. As mentioned earlier, the cluster discovery process can be improved by developing a new software clustering algorithm or developing a new resemblance coefficient technique. What is a better way to improve the cluster discovery process is an open question. Since this question is open, the research community is splitting its efforts by developing both software clustering algorithm and resemblance coefficient techniques instead of focusing on one or the other.

2. Several software clustering algorithms and resemblance coefficient techniques have been developed. Therefore, selecting a software clustering algorithm and a suitable resemblance coefficient is a challenging task. A comparative study tested on a number of systems is long overdue. It is possible that particular combinations of clustering algorithms with resemblance coefficients are better suited for a particular type of software system. A categorization of algorithms based on the types of software for which they work best would be beneficial to the software field.

3. The decomposition of a software system into subsystems is a typical task of a reverse engineer. A software clustering method can produce a nested decomposition; typically, based on the output of a hierarchical algorithm. Currently, there is no optimization-based software clustering algorithm that can produce a nested decomposition. We know that optimization algorithms can efficiently construct flat decompositions of a software system. It would be worth investigating whether an optimization algorithm would be an efficient algorithm for extracting a nested decomposition from a software system as well.

4. Since a particular software system may have various valid decompositions, the output of a software clustering algorithm is often not meaningful to a software engineer. To overcome this problem, some software clustering algorithms assign labels for each cluster. Label assignment is a challenging task and it is undeveloped in the context of software clustering. The problem is that software clustering algorithms either do not assign labels or assign labels based on simplistic rules. The develop-

ment of an advanced method for label assignment would be beneficial to the field.

Results Visualization

As presented earlier, the visualization of results is a challenging problem because often the results of software clustering methods are large decompositions. Therefore, it is complicated to develop a user interface, which is capable of presenting a software decomposition in a meaningful way to a software engineer. In addition, the user interface is supposed to allow navigation through software decomposition results and to link the decomposition to source files, documentation etc. Software clustering methods would be more useful if the visualization of the results of the software clustering were improved.

User Feedback

The importance of developing semi-automatic clustering algorithms was discussed earlier in this chapter. There are two main obstacles to developing advanced semi-automatic clustering algorithms. The first one is the lack of a user interface that allows the users to understand and evaluate the clustering results. The second, and more important obstacle is that the results of the semi-automatic algorithm may not reflect the current state of the software system because it is driven by the user whose knowledge of the software system is often incomplete. We think that it is important to develop criteria that allow the algorithm to disregard wrong user feedback. When such criteria have been established, then the research community will be able to pay more attention to semi-automatic software clustering algorithms.

Evaluation

Evaluation of software clustering algorithms has already been studied from various aspects, but still there are uncharted research areas including the following:

1. Each clustering approach uses a different set of evaluation measures from the several measurements that have been presented in the literature. Consequently, we cannot compare evaluation results from different studies. To overcome this problem, the evaluation metrics have to be compared to each other. It is worth investigating whether certain metrics are correlated with each other for particular types of software systems.

2. The research software community needs to define a standardized method of evaluating software clustering. This requires the collection of reference software systems. This collection has to contain many different types of software systems. The construction of such a collection is complicated because it is difficult to find good candidates. Most research studies propose using big open source software systems as reference systems. However typically, a big software system includes several types of software paradigms (event-based, web services etc). Another challenge is that new software types constantly appear. Until now, no study has developed a deep theoretic justification for the construction of a collection of reference systems. The reference collection would help to define a standard way of comparing software clustering systems. In addition, it would allow better exploration of properties of existing software clustering methods and development of new more advanced

methods.

3. Most of the research work presented in the literature is about the evaluation of the complete software clustering process. Until now, there is no dedicated study to establish a method for the evaluation of a specific phase of the software clustering process. Such a study would be an important tool for the research community. It would allow reverse engineers to select an appropriate algorithm for their task.

The remainder of this dissertation presents our approach to addressing several of the open questions related to the field of software clustering. Before doing so, we present the software systems that will be used in experiments throughout the thesis. We experimented with three large software systems. The particular versions of these systems were chosen because an authoritative decomposition existed for them.

- **TOBEY**. This is a proprietary industrial system that is under continuous development. It serves as the optimizing back-end for a number of IBM compiler products. The version we worked with was comprised of 939 source files and approximately 250,000 lines of code. The authoritative decomposition of TOBEY was obtained over a series of interviews with its developers.
- **Linux**. We experimented with version 2.0.27a of the kernel of this free operating system that is probably the most famous open-source system. This version had 955 source files and approximately 750,000 lines of code. The authoritative decomposition of this version of the Linux kernel was presented in [BHB99].

- **Mozilla**. This is a widely used open-source web browser. We experimented with version 1.3 that was released in March 2003. It contains approximately 4.5 million lines of C and C++ source code. This version had 3559 source files. A decomposition of the Mozilla source files for version M9 was presented in [GL00]. We used an updated decomposition for version 1.3 [Xia04].

In the next chapter we present tools that will facilitate our research.

3 Factbase and Decomposition Generators

The program comprehension research community has developed a large number of approaches that can help developers understand large software systems accurately and efficiently. Several of these approaches have corresponding implementations that are available online [bun, acd, swab]. However, tools that can facilitate research in program comprehension are rarely publicly available. This means that researchers have to recreate tools already developed by other community members, a process that requires significant amounts of time and energy.

While it is true that many research studies require dedicated tools, there are many general purpose tools that can be reused in different contexts. In this chapter, we introduce two such tools that generate artifacts that can be used to study the behaviour of existing approaches for the comprehension of large software systems:

- A factbase generator that produces simulated factbases that resemble factbases extracted from real software systems
- A decomposition generator that produces simulated decompositions of software systems into subsystems that resemble decompositions produced by system experts or existing clustering algorithms

Both presented tools support a number of configuration parameters that allow the user to produce output that is suited for their purposes. For example, the number of entities as well as an upper bound for the number of clusters in a software decomposition can be specified. Several experiments have been conducted to confirm that the two generator tools produce artifacts that adhere to the provided configuration parameters, as well as uniformly span the whole spectrum of possible artifacts.

We also present three distinct applications of the introduced tools: the development of a simple evaluation method for clustering algorithms, a study of the behaviour of the objective function of the Bunch tool [MMR+98], and the calculation of a congruity measure for clustering evaluation measures, such as MoJoFM [WT04b] and KE [KE00]. The congruity metric was used in several sets of experiments in Chapters 4 and 6 to determine whether MoJoFM and KE are correlated. The generator tools have also been used to study the comparability of software clustering algorithms (see Chapter 6).

All tools presented in this chapter are available online as part of the Java Reverse Engineering Toolkit (JRET) [jre].

The structure of the remainder of the chapter is as follows: The factbase generator tool is presented in Section 3.1. Section 3.2 introduces the decomposition generator tool. Experiments that demonstrate properties of the produced decompositions are presented in Section 3.3. Section 3.4 describes the application of the presented tools to three distinct research tasks and concludes the chapter.

3.1 Simulated Factbase Generator

Many of the program comprehension approaches published in the literature involve the extraction of relevant information from the system's source code into what is typically called a *factbase*, i.e. a collection of facts about the software system. Once extracted, a factbase can be used as input for a wide variety of program comprehension tasks, such as visualizing the extracted facts, clustering the system's entities, or mining for patterns.

However, the creation of such a factbase can be a time-consuming task. Research activities that require a large number of such factbases, such as performance and robustness testing for clustering or visualization tools, would benefit from the ability to create simulated factbases that closely resemble ones from real software systems. In this section, we present a method for the creation of module dependency graphs, the most common type of factbase.

A module dependency graph (MDG) has vertices that are system modules, such as functions, classes, or source files, and edges that represent dependencies between these modules, such as procedure calls or variable references. As a result, an MDG is a directed graph.

In order to create realistic simulated MDGs, we need to determine the relevant properties of MDGs from actual software systems. Our approach identifies a number of such properties, and our implementation [Sht] provides options to enable or disable any or all of them. The design of the implementation allows for the easy addition of further properties.

The graph properties identified by our approach range from simple ones, such as having no loops, to quite involved ones, such as the power law distribution for the node's degrees. The kind of dependency to be simulated by the produced MDG should determine which of the following properties should be enabled:

1. No multiple edges: Only one edge is allowed between two modules.

2. No loops: A module may not depend on itself.

3. No cycles: Certain types of dependencies, such as class inheritance, may not appear in a cycle, while others, like object reference, frequently contain cycles.

4. Connectivity: The produced graph needs to be connected. If the generation process described below produces a disconnected graph, our algorithm identifies the connected components and modifies as many edges as required in order to connect them (without violating other constraints).

5. Low Clustering Coefficient: The clustering coefficient of an undirected graph is a measure of the number of triangles in the graph [WS98]. The software graphs of Linux, Mozilla, and TOBEY have clustering coefficients that belong to the interval [0.2, 0.4], so we implemented a similar restriction for the simulated graphs. Since it is not practical to change the clustering coefficient of a graph by reshuffling edges, a violation of this constraint requires that the generation process has to be repeated.

6. Degree Power Law Distribution: Several studies argue that the degrees of the nodes in a typical MDG follow a power law distribution [Mye03, CMPS07, VS].

More precisely [GMY04], the probability $P(k)$ of a given node having an in- or out-degree k, is expressed as:

$$P(k) \sim k^{-\gamma},$$

where γ is a value larger than 1, called the power law exponent. The value of γ in large software systems is consistently found to be around 2.5 with the average in-degree exponent being slightly less than that and the average out-degree exponent slightly more [VS]. Our implementation uses 2.5 as the default value for the power law exponent.

The research tool presented in this section works in a manner similar to the following simplified process (adapted so it applies to a directed graph) from a typical algorithm for the generation of random graphs with a power law distribution for its degrees [ACL00]:

- Every node in the graph starts with a current in- and out- degree of 0.
- Every node in the graph is assigned a target in- and out-degree, so that the sequence of degrees follows a power law distribution. The sum of all target in-degrees is equal to the sum of all target out-degrees.
- Let I be the set of nodes whose current in-degree is less than their target in-degree. Let O be the set of nodes whose current out-degree is less than their target out-degree. While I and O are not empty, randomly select a node from each set. Create an edge between the two nodes.

In order to achieve the graph properties listed earlier, we modify the above process in order to apply additional constraints. We allow for two types of constraints:

1. Edge constraints. These are applied after each edge creation. If the constraint is violated, the edge creation is cancelled, and a new pair of nodes is selected. These ensure that the first three graph properties are satisfied: No multiple edges, no loops, and no cycles.

2. Graph constraints. These are applied after all edges have been created. If the constraint is violated, an effort will be made to satisfy it if possible by reshuffling some of the edges. If that is not possible, the graph generation process needs to be repeated.

Experiments using the factbase generator tool to study the comparability of software clustering algorithms (see Chapter 6) have demonstrated that the produced MDGs can be used successfully for research purposes.

3.2 Decomposition Generators

The second research tool introduced in this chapter allows a researcher to create simulated decompositions of large software systems. These can be used for a variety of tasks, such as studying the behaviour of software clustering evaluation measures [WT04b, KE00, MM01a], or testing the performance and robustness of visualization tools [Sof].

Section 3.2.1 presents two versions of an algorithm for the generation of simulated flat software decompositions, while Section 3.2.2 presents such an algorithm for nested

decompositions.

3.2.1 Flat Decomposition Generators

The flat decomposition generation algorithm requires two inputs:

1. E: The number of entities to be clustered. Alternatively, a list of entity names can be provided. This way the produced decomposition will contain entities whose names are meaningful to the researcher.

2. U: An upper bound for the number of clusters in the produced decomposition. The exact number of clusters A will be a random number from the interval [2,U].

Once the number of clusters is determined, the generation algorithm proceeds in one of two ways:

- Unrestricted cardinality: In this version, each entity is randomly assigned to one of the clusters. Any clusters that remain empty after all entities have been assigned are deleted from the produced decomposition.

- Predetermined cardinality: In this version, each cluster is randomly assigned a cardinality before any entity distribution takes place. When entities are assigned to clusters, a cluster is selected only if its current size is less than the assigned cardinality.

  The sum of all cardinalities is, of course, equal to the number of entities to be clustered. Also, in order to avoid creating unrealistic software decompositions, 70% of

the cardinalities are chosen from a normal distribution around the expected cardinality $\frac{E}{A}$, while the rest are random numbers from the interval [1,E-1].

As will be shown in Section 3.3, both versions of the generation algorithm can produce all possible software decompositions for a given E and U. However, the two versions produce decompositions that differ in structure in the average case. Depending on the relation between E and A, they may produce balanced or unbalanced decompositions. Table 3.1 presents all possibilities.

Algorithm	Inputs	Decomposition
Unrestricted Cardinality	$A \ll E$	Balanced
Unrestricted Cardinality	$A \sim E$	Unbalanced
Predetermined Cardinality	$A \ll E$	Unbalanced
Predetermined Cardinality	$A \sim E$	Balanced

Table 3.1: The relation between algorithm version, inputs, and output decomposition type.

A researcher should use the information in Table 3.1 as a general guideline when selecting input values as well as algorithm version for their research task.

3.2.2 Nested Decomposition Generator

The generation algorithm for simulated nested decompositions is in fact a simulation of a divisive hierarchical clustering algorithm, i.e. it begins by placing all entities in one

cluster, which is then divided into successively smaller clusters. In order to achieve this, our algorithm requires three inputs:

1. E: The number of entities to be clustered, or a list of entity names as in the case of the flat generator.
2. C: An upper bound for the number of subclusters for a given cluster.
3. D: An upper bound for the height of the containment tree.

The generation algorithm starts by creating a flat decomposition using one of the algorithms presented in the previous section (U is equal to C in this case). This becomes the first level of the nested decomposition. An upper bound L for the height of the containment subtree rooted at each first level cluster is determined as a random number from the interval [1,D].

For each one of the clusters of the first level, the following process is repeated iteratively:

Create a flat decomposition for the entities in the cluster. Update the value of L as a random number from [0,L-1]. Stop if L=0, otherwise iterate to the next level.

The algorithm is presented in pseudocode in Figure 3.1.

In the next section, we present experiments that attempt to determine the randomness of the decompositions produced by the algorithms presented above.

```
GenerateNestedDecomposition(E,C,D:int)
  begin
    F := GenerateFlatDecomposition(E,C)
    L := Random number in [1,D]
    Assign L as subtree height to all clusters in F
    Create empty nested decomposition N
    Assign F as the first level of N
    while there exists cluster x in N whose subtree height is
           larger than 0 do
      F := GenerateFlatDecomposition(|x|,C)
      L := Random number in [0,L-1]
      Assign L as subtree height to all clusters in F
      Replace cluster x with F
    end while
  end
```

Figure 3.1: Pseudocode for the simulated nested decomposition generation process.

3.3 Decomposition Generator Properties

For the research tools presented in this chapter to be useful, the generated factbases and decompositions would need to span the entire space of possible outputs. Our factbase generator is based on an algorithm that has been shown to have this property [ACL00]. As a result, in this section, we focus on investigating whether the decomposition generators presented in Section 3.2 produce decompositions with the following two properties:

- **Unique**: The probability that two generated decompositions are equal must be very small. Our experimental results are presented in Section 3.3.1.
- **Random**: To our knowledge, no test for the randomness of decompositions exists currently. We adapt methods developed in the context of cryptography [RoSU00] to validate random number generators for our purposes. Our experimental results are presented in Section 3.3.2.

3.3.1 Uniqueness

In this section, we investigate the uniqueness of the decompositions created by the generation algorithms presented earlier. We present experiments using the nested decomposition generator, since a flat decomposition is a special case of a nested decomposition. Experiments using the flat generators have yielded similar results.

An important issue to consider is that of isomorphic decompositions. Figure 3.2 presents two different representations for the same nested decomposition. To address this problem, we applied a method developed for comparing unsorted trees [Bus97]. We define a

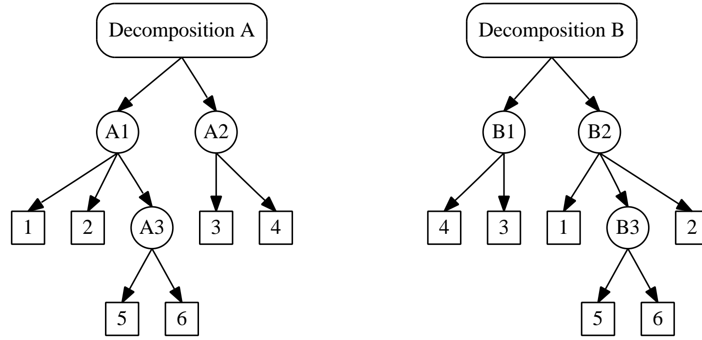


Figure 3.2: An example of two isomorphic nested decompositions.

string representation of a nested decomposition in such a way, so that two nested decompositions are equal iff their string representation is identical.

Definition 3.3.1. The **string representation** $R(T)$ of nested decomposition T is a string constructed as follows:

1. If $|T| = 1$, then $R(T)$ equals `"{root}"`, where `root` is the label of the single entity in the decomposition.

2. If $|T| > 1$, then let $S_1, ..., S_m$ be the immediate subtrees of T ordered so that $S_1 \succeq S_2 \succeq ... \succeq S_m$, where $\succeq$ is any partial order. Then, $R(T)$ is equal to "$\{R(S_1)R(S_2)...R(S_m)\}$".

For example, `"{{{1}{2}{{5}{6}}}{{3}{4}}}"` is the string representation for the two decompositions shown in Figure 3.2, when the partial order used is lexicographical sorting.

In order to assess the uniqueness of the decompositions produced by our tool, we created a large number of nested decompositions, constructed the string representation for each one, and computed the number of duplicate string representations.

We performed 100 experiments with all described generators. In every experiment, 1000 decompositions were generated. The input parameters were: E = 1000, C = 10, D = 9. There was only one pair of duplicate decompositions found in all these experiments. This confirms that our algorithm produces unique decompositions.

3.3.2 Randomness

Next, we investigate whether the decomposition generators produce well randomized sets of decompositions. We concentrated on the following variables and identified a null hypothesis for the frequency distribution of each variable:

- Number of clusters in a flat decomposition. Null hypothesis: Uniform distribution in the range of requested values.
- The probability that two entities belong in the same cluster in a flat decomposition. Null hypothesis: This probability is the same for each pair of entities.
- Number of children for a non-leaf node of a nested decomposition. Null hypothesis: Uniform distribution in the range of requested values.
- The probability that two entities belong in the same cluster in a nested decomposition. Null hypothesis: This probability is the same for each pair of entities.
- Height of leaf nodes in a nested decomposition. Null hypothesis: Uniform distribution in the range of requested values.

We used the chi-square (χ^2) test statistic [MGB74] to measure how well the observed frequency distribution for the above variables matches the expected one. For an experiment with k possible outcomes, the value of the chi-square test statistic is given by the formula:

$$\chi^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i}$$

where E_i is an expected frequency asserted by the null hypothesis, and O_i is the observed frequency.

Based on the value of the chi-square statistic, one can calculate the p-value, i.e. the probability that the differences between the observed and expected frequencies occurred by chance. If that probability is small (typically less than 0.05) then the null hypothesis is rejected.

In our experiments, the p-values we observed where always in the interval [0.9, 1], even though we repeated each experiment 1000 times. While this is not equivalent to proving the null hypotheses, it provides a strong indication that the decomposition generators presented in Section 3.2 are well randomized.

3.4 Applications

We now present three distinct situations where the application of the generator tools presented in this chapter can help provide significant insight in various open questions related to software clustering.

3.4.1 A simple clustering evaluation method

Software clustering algorithms decompose a software system into subsystems. To evaluate the quality of a software clustering algorithm, one typically compares the decomposition produced by a software clustering algorithm with a decomposition produced by an expert. However, due to the complexity of most real industrial systems, experts often disagree about what constitutes a good decomposition. This situation makes evaluation of software clustering algorithms a rather difficult task.

For the clustering evaluation method presented in this section, we make the assumption that when the structure of a software system is not complex, then experts and clustering algorithms alike should agree on how to decompose the system into subsystems. For instance, if the MDG of a software system is a rooted tree, such as the one presented in Figure 3.3, one would expect that a decomposition such as the one shown in Figure 3.4 would be universally accepted[3].

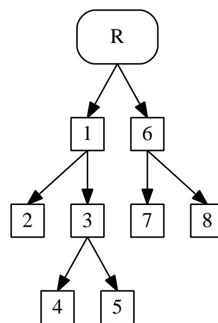


Figure 3.3: A rooted tree MDG.

By utilizing the factbase generator tool presented in Section 3.1, we can devise a simple

[3]The root entity can be assigned to any cluster from the universally accepted decomposition. In our experiments, we consider the placement of the root to be correct regardless of the cluster it belongs to.

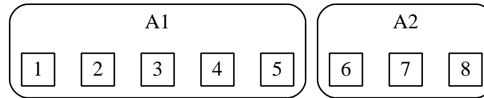


Figure 3.4: The decomposition that should correspond to the MDG in Figure 3.3.

way to evaluate how well clustering algorithms, such as ACDC and Bunch, perform in such a situation. Our evaluation method proceeds as follows:

- Construct an MDG that is a rooted tree by requiring no loops, no cycles, and a connected graph.
- Construct a decomposition from the MDG using the clustering algorithm being evaluated.
- Calculate the MoJoFM distance between the expected decomposition and the decomposition constructed by the clustering algorithm.
- Repeat from the beginning a large number of times.

We applied this evaluation measure to ACDC and Bunch by repeating the above process 100 times for three different sets of input values. The obtained results are shown in Table 3.2.

The results show that even established software clustering algorithms, such as ACDC and Bunch, may fail to discover good software decompositions from software systems with trivial structure. The MDGs created during these experiments may provide interesting insight as to why the algorithms deviated from the expected result.

E	C	D	Algorithm	MoJoFM values	Mean	Standard Deviation
1000	20	10	ACDC	60.73 - 100	88.91	9.4
			Bunch	33.88 - 100	77.49	18.06
800	20	10	ACDC	76.29 - 100	90.87	6.62
			Bunch	34.25 - 100	79.77	17.16
800	10	10	ACDC	91.3 - 100	95.57	4.77
			Bunch	51.25 - 100	83.77	14.16

Table 3.2: Results for the simple clustering evaluation method.

It is also important to note that the evaluation method just presented does not require the existence of an authoritative decomposition. Very few clustering evaluation methods share this property [TH00b, WHH05].

3.4.2 Properties of Bunch's MQ function

For our next application, we will use both presented generator tools in order to study the properties of the objective function used by Bunch. It is well-known that, due to the fact that it uses randomization, Bunch may create different results for subsequent runs using the same input parameters. We wish to investigate whether this can be attributed to the fact that the objective function MQ has a large number of local optima.

The way Bunch clusters software systems is as follows: Start with an initial decomposition that may be created randomly or provided by the user. Modify the decomposition by moving entities between clusters. If these changes improve the value of the objec-

tive function MQ, then accept the changes, otherwise discard them. Repeat this process until no further improvement can be made to the value of MQ. It has been shown that Bunch will always find a decomposition whose MQ values is close to the global maximum [Xan06].

We run the following experiment:

1. Create a simulated factbase F using the generator tool presented in Section 3.1. For the experiments presented the following constraints were activated: no multiple edges and no loops. The number of entities (E) was 2000.

2. Create decomposition B_0 by running Bunch with F as input.

3. Construct 100 simulated decompositions S_i ($1 \leq i \leq 100$) that refer to the same entities as F by using the decomposition generator tools in Section 3.2.1. The input values used were: E = 2000, U = $|B_0|$ + 10. The value of parameter U was selected so that the simulated decompositions have a cardinality that is similar to B_0.

4. Construct decompositions B_i ($1 \leq i \leq 100$) by running Bunch with F as input and S_i as the initial decomposition.

5. Compute the MQ value for all B_i, and compare to MQ(B_0).

We repeated this experiment 10 times. Between 20% to 30% of the B_i decompositions had an MQ value higher than that of B_0. This shows that there are several distinct decompositions that have a high MQ value, which implies the existence of a large number of local optima. It is interesting to note that the B_i decompositions that had higher MQ

values than B_0 were distinctly different from B_0. The MoJoFM similarity between B_i, for which MQ(B_i) $>$ MQ(B_0), and B_0 ranged in the interval [35.9, 45.65].

3.4.3 Congruity of Clustering Evaluation Measures

The last application of the generator tools introduced in this chapter has to do with investigating the behaviour of clustering evaluation measures, such as MoJoFM and KE. While the goal of such measures is the same – to determine the similarity between two different decompositions of the same software system – they often lead to contradictory results. (see Chapter 5).

Suppose we want to compare three decompositions `A`, `B`, and `C` of the same software system using evaluation measures M and U (the higher the value of M or U, the more similar the decompositions). By applying both measures to all pairs of decompositions, we obtain the results in Table 3.3.

Decomposition pair	(`A`,`B`)	(`B`,`C`)	(`A`,`C`)
M values	3	2	1
U values	5	8	2

Table 3.3: Evaluation measure values for all decomposition pairs.

According to evaluation measure M, `A` and `B` are the two most similar decompositions. However, according to U, this is incorrect (`B` and `C` are the two most similar). This is a significant difference because it affects the way the two algorithms rank the three decompositions. In other words, we would not consider it a noteworthy issue if U produced

larger similarity values than M, as long as the two algorithms ranked all decompositions in the same order of similarity.

A generalization of this idea leads to a congruity metric that determines correlation between two evaluation methods of software clustering algorithms. Assume we have N pairs of decompositions of the same software system. We can apply both M and U to each pair and obtain two different values m_i and u_i. We arrange the pairs of decompositions in such a way so that for $1 \leq i \leq N-1$, we have $m_i \leq m_{i+1}$. The value of the congruity metric will be the number of distinct values of i for which $u_i > u_{i+1}$. Values for this metric range from 0 to $N-1$. A value of 0 means that both comparison methods rank all pairs in exactly the same order, while a value of $N-1$ probably indicates that we are comparing a distance measure to a similarity measure. Values significantly removed from both 0 and $N-1$ indicate important differences between the two comparison methods.

The calculation of the congruity metric requires a large number of pairs of software decompositions. Using the tools introduced in this chapter, we can devise two distinct ways of generating these decompositions:

The first generation method utilizes any of the decomposition generator algorithms presented in Section 3.2.

The second generation method starts by generating a random factbase using the process described in Section 3.1. Next, two different clustering algorithms construct two distinct software decompositions based on the same factbase.

Both generation methods have advantages and disadvantages. The main drawback of the first method is that the decompositions may be significantly different from each other.

This situation is unlikely when the two decompositions are actual clustering results. The main drawback of the second method is its dependency on the two software clustering algorithms used. Experiments with several pairs of algorithms may have to be conducted in order to remove possible bias in the obtained results.

We will use the congruity metric presented in this section in several sets of experiments in Chapter 4 and 6 to determine whether MoJoFM and KE are correlated in different contexts.

In the next chapter, we start tackling the problem of evaluating nested decompositions.

4 Evaluation of nested decompositions

Retrieving design information from the source of a software system is an important problem that affects all stages of the life cycle of a software system. Usually, complicated systems consist of different subsystems that can help divide such a system into logical components. The natural decomposition of a software system is usually presented as a nested decomposition [HB85]. Many different methodologies and approaches that attempt to create such decompositions automatically have been presented the literature[AT03, CS90, HB85, MMCG99, MU90, TH00a]. Most of them produce nested decompositions.

Evaluating the effectiveness of software clustering approaches is a challenging issue. Comparing results produced by different tools is a complicated problem. A number of approaches that attempt to tackle this problem have been presented [KE00, MM01a, TH99]. However, all of them assume a flat decomposition.

In this chapter, we present a framework called END that allows one to reuse a technique developed for flat decompositions in order to compare nested ones. The END framework takes into account all levels of a nested decomposition when making the comparison. We apply several existing comparison methods to the END framework, and we use it to compare nested decompositions for two large software systems. Our experi-

ments show that the END framework can provide more accurate results than the original methods.

In addition, we present a new approach to the comparison of nested decompositions. This approach, called UpMoJo, is a generalization of the MoJo distance measure for flat decompositions [TH99]. UpMoJo adds an Up operation to the existing Move and Join operations of MoJo which allows for differences in the hierarchical structure of two different decompositions of the same system to be reflected in the distance measured.

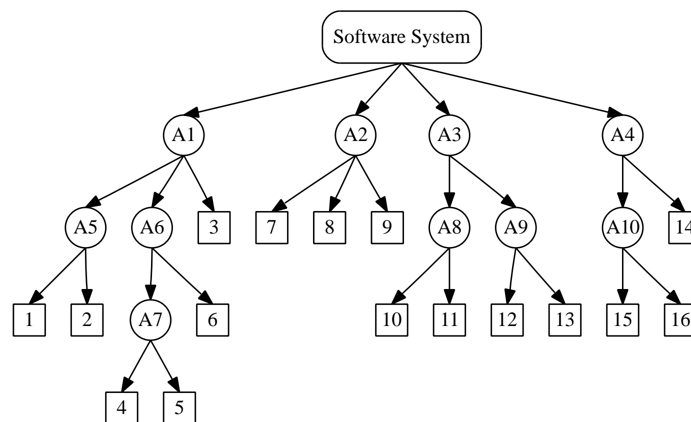
The structure of the rest of this chapter is as follows. Section 4.1 presents the issues that arise when nested decompositions are flattened in order to be compared using existing methods. Our solution to this problem, the END framework, is presented in Section 4.2 along with experiments that showcase the usefulness of the framework. Section 4.3 presents the motivation for UpMoJo. The UpMoJo distance measure and related experiments are introduced in Section 4.4.

4.1 Flat vs. nested decompositions

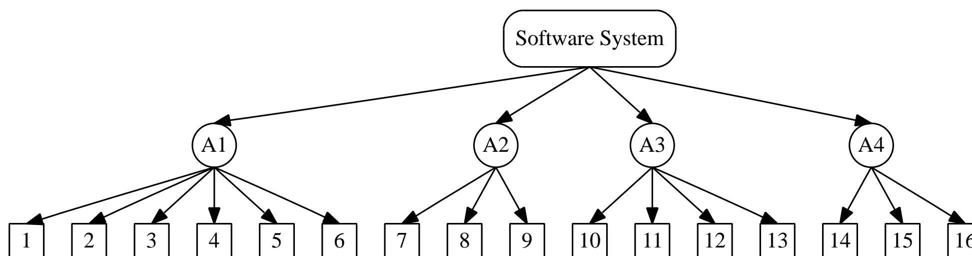
The clustering evaluation methods presented in Section 2.3, such as MoJoFM or KE, have been developed in order to compare flat decompositions. Since clustering algorithms commonly create nested decompositions, various methods have been devised in order to evaluate clustering results using these methods. The most common approach is to convert the nested decompositions into flat ones before applying the comparison method. However, this approach has limitations.

Converting a nested decomposition to a flat one is commonly done in two different ways depending on whether a compact or a detailed flat decomposition is required:

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. The flat decomposition obtained contains only the top-level clusters of the original one (Figure 4.1 presents an example of such a conversion).



(a) A nested decomposition of a software system



(b) The compact flat form of decomposition (a)

Figure 4.1: Conversion of a nested decomposition to a compact flat decomposition.

2. Converting a nested decomposition to a detailed flat one. Each cluster and its subtree is assigned directly to the root of the containment tree. The decomposition obtained contains any cluster that contained at least one object in the original decomposition. Figure 4.2 presents the detailed flat form of the decomposition in Figure 4.1(a).

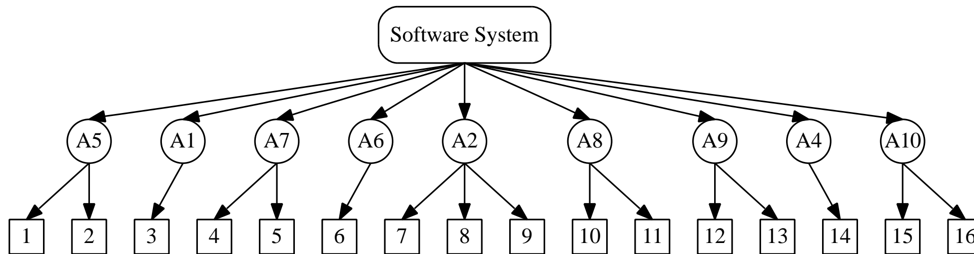


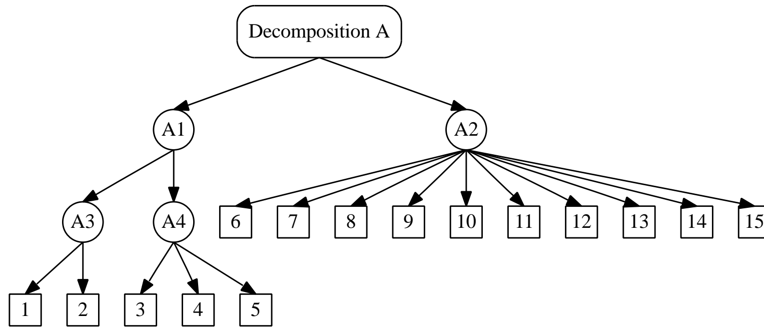
Figure 4.2: The detailed flat form of the decomposition in Figure 4.1(a).

Figure 4.3 presents three decompositions of the same software system. Clearly, decompositions (a) and (b) have different hierarchical structures. Such decompositions cannot be compared directly using one of the methods presented already.

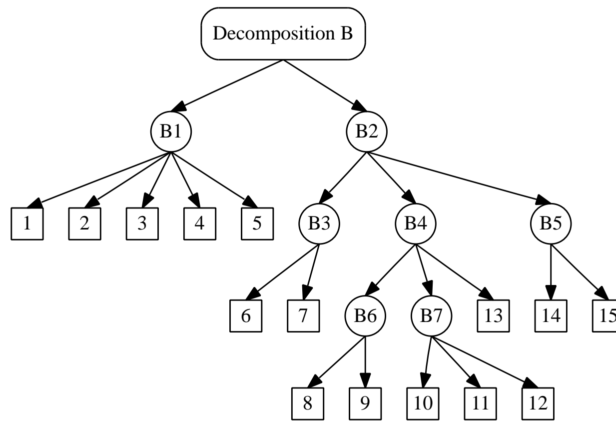
If we convert the decompositions in Figure 4.3 to compact flat form, they will both become equivalent to decomposition (c). However, the original decompositions were quite different. A significant amount of information has been lost. Figure 4.4 presents an example where detailed transformation creates a similar problem.

These examples illustrate clearly that converting nested decompositions to flat ones removes significant information that could impact the evaluation process.

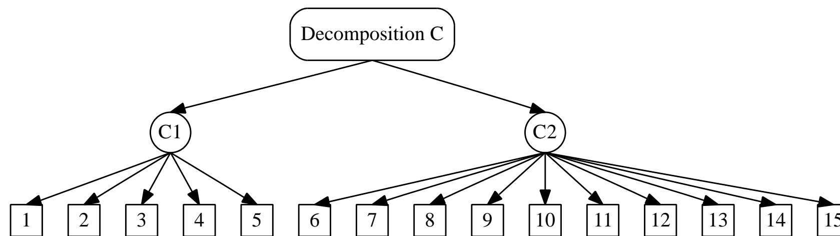
The next section discusses a framework that alleviates this problem.



(a) A nested decomposition of a software system

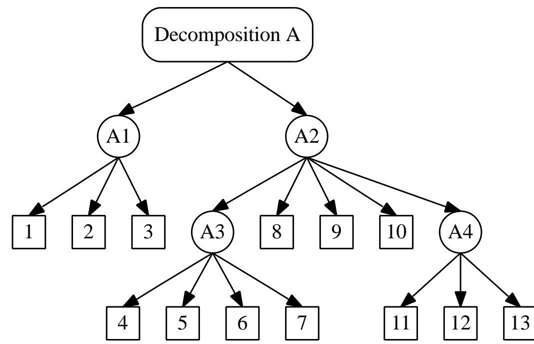


(b) Another nested decomposition of the same software system

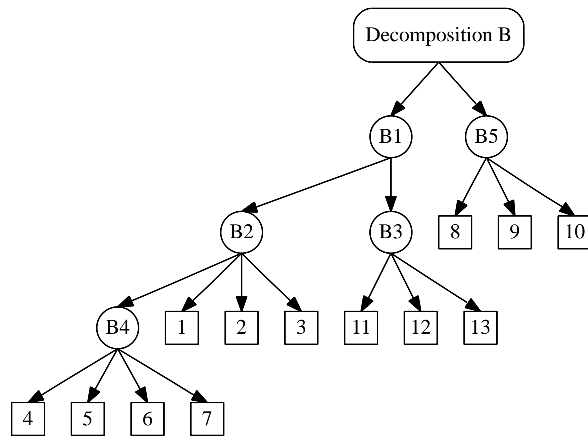


(c) The compact flat form of both decompositions (a) and (b)

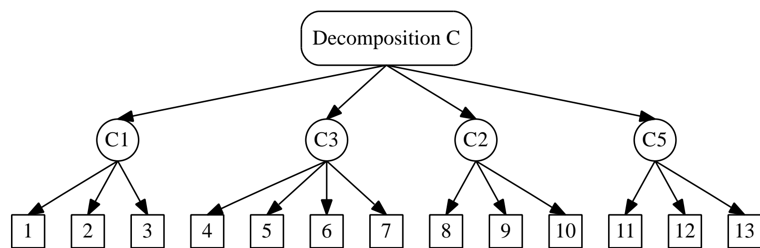
Figure 4.3: Limitation of compact flat decompositions.



(a) A nested decomposition of a software system



(b) Another nested decomposition of the same software system



(c) The detailed flat form of both decompositions (a) and (b)

Figure 4.4: Limitation of detailed flat decompositions.

4.2 The Evaluation of Nested Decompositions (END) framework

The Evaluation of Nested Decompositions (END) framework is able to compare two nested decompositions without any information loss by converting them into vectors of flat decompositions and applying existing comparison methods to selected elements of these vectors. Details of the process are presented in Section 4.2.1. Experiments using the END framework are presented in Section 4.2.2.

4.2.1 END framework

The presentation of the END framework begins by introducing certain terms that will help our discussion.

Each cluster in the nested decomposition is assigned a *level* that is equal to the length of the path from the root of the decomposition to itself. For example, in Figure 4.3b, cluster `B2` has a level of 1, while cluster `B7` has a level of 3. The *height* of a nested decomposition is defined as its maximum cluster level.

The algorithm that transforms a nested decomposition to a vector of flat ones works as follows:

For each cluster level ℓ in the nested decomposition we construct a flat decomposition as described below:

1. Each object in a cluster of level larger than ℓ is assigned to its ancestor of level ℓ. This results in a nested decomposition T of height ℓ.

2. Convert T to a detailed flat decomposition, as described in Section 4.1.

We denote a flat decomposition corresponding to level ℓ with P_ℓ.

Figure 4.5 shows P_2 for the decomposition in Figure 4.3b.

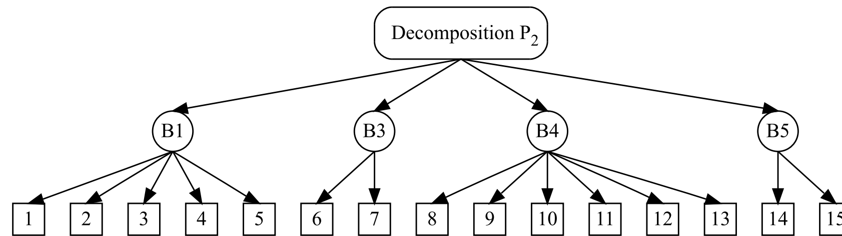


Figure 4.5: Flat Decomposition P_2 for the decomposition in Figure 4.3b.

Definition 4.2.1. Nested decomposition vector (NDV) of a nested decomposition is a vector, where NDV_i=P_i.

The size of an NDV is equal to the height of its corresponding nested decomposition. It is interesting to note that the first element in the NDV is a compact flat decomposition, while the detailed flat decomposition is its last element.

Our method for the comparison of two nested decompositions depends on the fact that a technique to compare flat decompositions already exists. Let us call this technique *M*. Consider two nested decomposition vectors V and U constructed from different software decompositions of the same software system. If the two vectors have different sizes, then the shorter one is expanded so that the two sizes are equal. Let us assume without loss of generality, that V is shorter than U. If the size of V is s, then all new elements appended to V are equal to V_s. In other words, any extra flat decompositions appended to V are equal to the detailed flat decomposition.

Definition 4.2.2. Similarity vector S_M of two nested decompositions with NDVs V and U is a vector of equal size to V and U, where $S_{M_i} = M(V_i, U_i)$.

As a result, the similarity vector S_M contains the value of the comparison method M for all cluster levels.

The overall similarity between the two nested decompositions will be computed based on the similarity vector. There are many ways to convert the similarity vector to a number, such as taking the norm of the vector. A more flexible approach would allow different weights for different coordinates. The formula would be the following:

$$\sqrt{\sum_{i=1}^{N}(w_i {S_{M_i}}^2)} \tag{4.1}$$

where $\sum_{i=1}^{N} w_i = 1$ and $N = |S_M|$ so that the overall result is normalized and can be compared to numbers obtained without using the END framework.

It is interesting to note that if the vector of weights is (1,0,...,0), then END becomes equivalent to comparing flat compact decompositions, while if the vector is (0,...,0,1), then END is equivalent to comparing flat detailed decompositions. In this sense, the END framework generalizes existing approaches, while providing more flexibility to the reverse engineer.

In the next section, we present several experiments with an implementation of the END framework. We have created plugins for several of the comparison methods presented in Section 2.3. We present results for MoJo, MoJoFM, KE, EdgeSim, and EdgeMoJo.

4.2.2 Experiments

The goal of the experiments presented in this section is to show that END can be useful to software clustering researchers. We present two different types of experiments. The target of the first series of experiments is to compare the results obtained from existing comparison methods to results obtained by the END framework using the same comparison method as a plug-in. The target of the second series of experiments is to demonstrate the capability of END to expose behaviour properties of flat evaluation measures.

The experiments were run on nested decompositions of two large open source software systems, Linux and Mozilla. Apart from the authoritative decompositions of these two systems, we also created nested decompositions by clustering them with two well-known software clustering algorithms, ACDC and Bunch.

Table 4.1 summarizes the decompositions used in the following experiments. It also presents the abbreviations we will use for them in this section, as well as their height.

We conducted two different types of experiments as outlined in the next two sections.

END vs Existing Approaches

The target of the first series of experiments is to compare the results obtained from existing comparison methods to the results obtained by the END framework using the same measurement as a plug-in. We calculated the (dis)similarity between different nested decompositions using both the END framework and the standalone approaches. For each standalone approach, we calculated two values: the value obtained when using the com-

Sign	Description	Height
L	Linux Authoritative Decomposition	6
LA	Linux ACDC Decomposition	3
LB	Linux Bunch Decomposition	4
M	Mozilla Authoritative Decomposition	5
MA	Mozilla ACDC Decomposition	4
MB	Mozilla Bunch Decomposition	5

Table 4.1: Nested decompositions used in the experiments.

pact flat decomposition, and the value obtained when using the detailed one. In this series of experiments, END is using the same weight for all components of the similarity vector.

Table 4.2 presents a summary of the obtained results. A graphical representation of the results is also given in Figure 4.6. In all cases, a lower value indicates that the two decompositions were more similar[4]. Note that values for the KE measure range from 0 to 1, MoJoFM and EdgeSim values range from 0 to 100, while MoJo and EdgeMoJo values are positive and bounded by s and 2s respectively, where s is the number of elements in each decomposition.

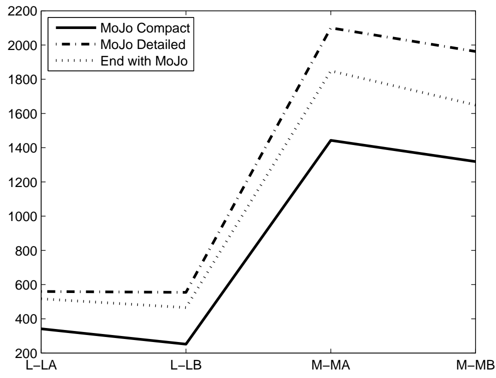
A number of interesting observations can be made based on these results.

To begin with, it is noteworthy that if one assumes balanced decompositions in terms

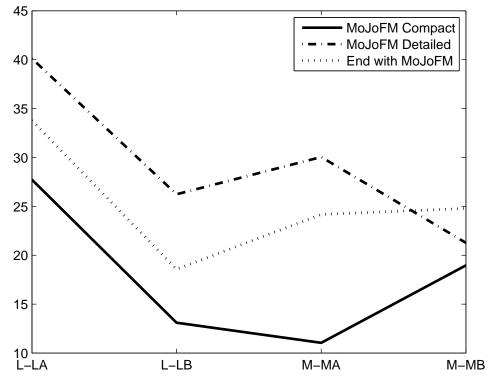
[4]For uniformity of presentation, we used customized versions of MoJoFM, KE and EdgeSim. The customized version returns a value that is the complement of the value returned by the original version. For example, a MoJoFM value of 65% is presented here as $100 - 65 = 35\%$.

Method	L - LA	L - LB	M - MA	M - MB
MoJo C	342	252	1443	1319
MoJo D	560	555	2100	1962
MoJo E	517.67	466.02	1848.80	1649.37
MoJoFM C	27.73	13.11	11.05	18.97
MoJoFM D	40.11	26.64	30.04	21.28
MoJoFM E	33.78	18.57	24.17	24.80
KE C	0.07	0.14	0.06	0.08
KE D	0.36	0.15	0.17	0.12
KE E	0.23	0.12	0.15	0.12
EdgeSim C	55.66	74.24	59.69	71.68
EdgeSim D	70.75	86.17	68.88	76.37
EdgeSim E	64.55	80.46	65.80	74.66
EdgeMoJo C	1317.10	1581.56	6193.23	5336.21
EdgeMoJo D	987.88	1331.72	4303.46	5387.47
EdgeMoJo E	1161.72	1508.87	5066.31	4881.42

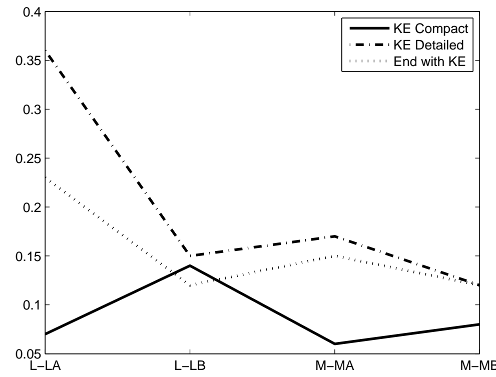
Table 4.2: Summary of Comparing Nested Decompositions (C = compact, D = detailed, E = END).



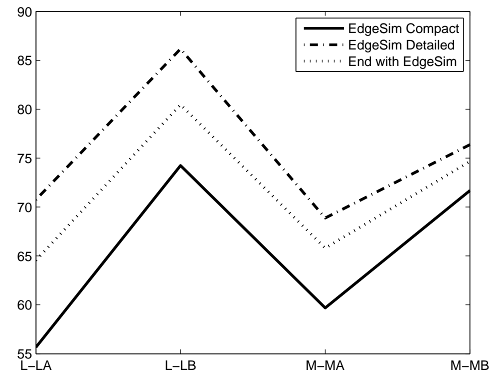
(a) MoJo



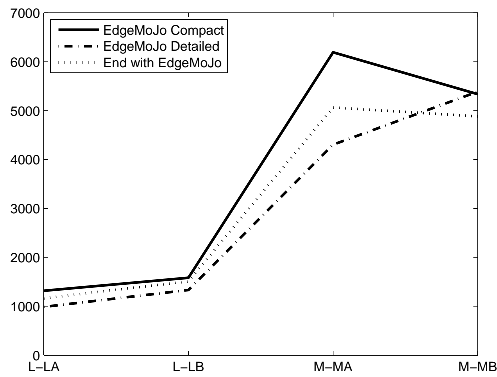
(b) MoJoFM



(c) KE



(d) EdgeSim



(e) EdgeMoJo

Figure 4.6: Comparing END to various standalone approaches.

of the number of entities per cluster, as well as the number of clusters per level, the expected order of the three plots in each graph would be: the plot corresponding to the detailed decomposition should be at the top, followed by the one corresponding to the END framework, while the plot corresponding to the compact decomposition should be the lowest. As can be seen in Figure 4.6, MoJo and EdgeSim follow this pattern. Any deviations from this - usually indicated by a crossing between the plots - are worthy of discussion.

For example, in the MoJoFM graph the “END plot” is higher than the “detailed plot” for the data point that corresponds to Mozilla and Bunch. This indicates that considering the intermediate levels as well helps Bunch achieve a better result. In other words, it appears that Bunch selects good top-level clusters, but does not necessarily do as well for the fine-grained ones. Since Bunch provides an option to create a decomposition at any level requested by the user, our experiments would indicate that selecting a higher level is recommended.

Another interesting observation is that the expected order for EdgeMoJo is actually reversed. This indicates that the edge component of EdgeMoJo penalizes more at the compact decomposition level. This can be explained by the fact that at that level, a *Move* operation between two clusters would result in the displacement of a large number of edges. This interesting property of the EdgeMoJo metric was revealed through the use of the END framework for the first time.

The results for the KE measure corroborate our previous observation for Bunch, since the END framework and the detailed decomposition produce the same value for Mozilla.

However, this time we see a different phenomenon for Linux. The compact and detailed plots are far apart for ACDC, but they are almost equal for Bunch. This is exactly the motivation for this work. Concentrating on a particular flat decomposition can provide biased results. Considering all levels (possibly with different weights) can give a better view of the quality of a particular decomposition.

Finally, it is interesting to note that in all experiments the values obtained for ACDC conformed to the expected order (with the already explained exception for EdgeMoJo). This is probably due to the fact that ACDC creates clusters of bounded cardinality, a restriction that does not apply to Bunch.

Various weighting schemes

As explained above, the END framework does not specify exact rules for the conversion of the similarity vector to a number. This can be done in a way that fits the specific needs of the reverse engineer using the framework. The previous experiments used equal weights for all elements in the similarity vector. However, other weighting schemes are also possible. We present and experiment with five of them:

1. The first element of the similarity vector has larger weight than the rest, i.e. more importance is assigned to the compact flat decomposition. We denote this weighting scheme by F.

2. The last element of the similarity vector has larger weight than the rest, i.e. more importance is assigned to the detailed flat decomposition. We denote this weighting

scheme by L.

3. All elements of the similarity vector have the same weight (denoted by ALL).

4. Each element in the vector has a weight equal to its index, i.e. the first element has a weight of 1, the second a weight of 2 etc. These weights are of course normalized before they are applied. We denote this scheme by UP.

5. Similar to UP, but in reverse order, i.e. the last element has a weight of 1, the second last a weight of 2 etc. (denoted by DOWN).

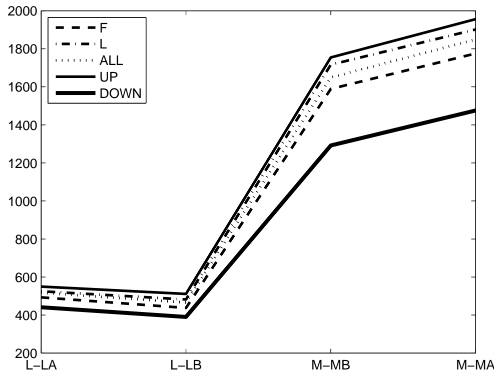
The actual formulas that were used in each case are given in Table 4.3. For simplicity, in these formulas $x_i = S_{M_i}$.

Abbr	Formula
F	$\sqrt(\sum_{i=2}^{N}(x_i^2\frac{1}{N+1}) + x_1^2\frac{2}{N+1})$
L	$\sqrt(\sum_{i=1}^{N-1}(x_i^2\frac{1}{N+1}) + x_N^2\frac{2}{N+1})$
ALL	$\sqrt(\sum_{i=1}^{N}(x_i^2\frac{1}{N}))$
UP	$\sqrt(\sum_{i=1}^{N}(x_i^2\frac{2i}{N(N+1)}))$
DOWN	$\sqrt(\sum_{i=1}^{N}(x_i^2\frac{2(N-i+1)}{N(N+1)}))$

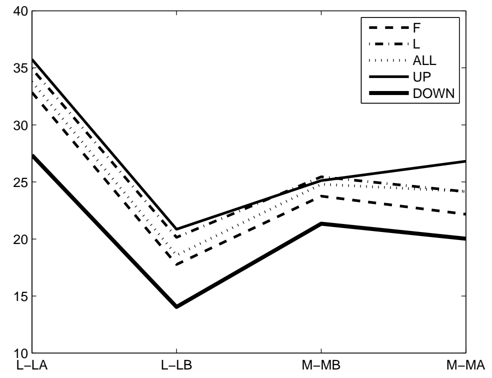
Table 4.3: Formulas for all weighting schemes.

Table 4.4 presents the results of these experiments. They are also presented in graphical form in Figure 4.7.

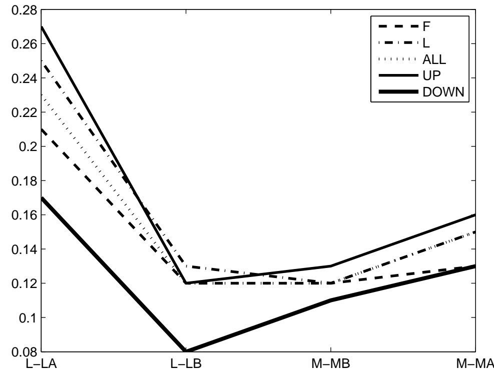
Assuming balanced decompositions, the expected order for the various plots in these graphs is the following (in order from top to bottom): UP, L, ALL, F, DOWN. As can be



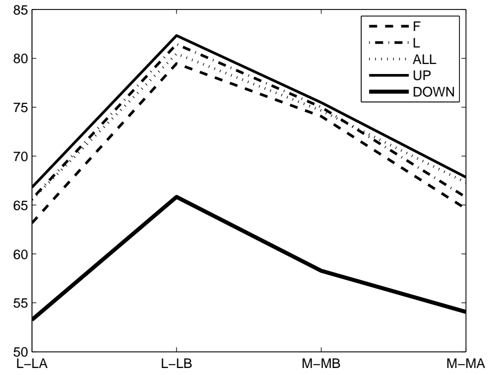
(a) MoJo



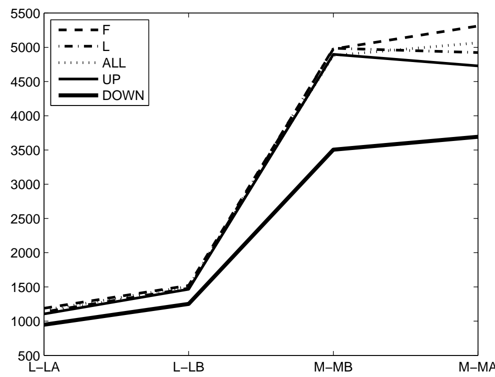
(b) MoJoFM



(c) KE



(d) EdgeSim



(e) EdgeMoJo

Figure 4.7: Weighting scheme results.

MoJo	L - LA	L - LB	M - MA	M - MB
F	492.76	437.68	1775.08	1588.80
L	524.97	481.99	1901.70	1716.46
ALL	517.67	466.02	1848.80	1649.37
UP	549.78	510.87	1955.81	1754.17
DOWN	441.27	390.12	1475.81	1291.45
MoJoFM	L - LA	L - LB	M - MA	M - MB
F	32.85	17.77	22.17	23.75
L	34.91	20.14	24.14	25.45
ALL	33.78	18.57	24.17	24.80
UP	35.74	20.85	26.81	25.12
DOWN	27.35	14.05	20.02	21.34
KE	L - LA	L - LB	M - MA	M - MB
F	0.21	0.12	0.13	0.12
L	0.25	0.13	0.15	0.12
ALL	0.23	0.12	0.15	0.12
UP	0.27	0.12	0.16	0.13
DOWN	0.17	0.08	0.13	0.11
EdgeSim	L - LA	L - LB	M - MA	M - MB
F	63.16	79.46	64.62	74.07
L	65.63	81.44	65.80	75.01
ALL	65.55	80.46	67.30	74.66
UP	66.81	82.33	67.84	75.48
DOWN	53.27	65.83	54.07	58.28
EdgeMoJo	L - LA	L - LB	M - MA	M - MB
F	1189.02	1521.23	5310.86	4975.70
L	1134.59	1480.82	4923.21	4986.74
ALL	1161.72	1508.87	5066.31	4881.42
UP	1105.45	1466.79	4727.89	4897.23
DOWN	948.11	1252.26	3693.19	3504.79

Table 4.4: Experimental results with various weighting schemes.

seen in Figure 4.7, the values for MoJo follow this pattern. As before, crossings in these graphs make for interesting observations.

For instance, in the MoJoFM graph, we have that UP is lower than L in the case of Mozilla and Bunch. This means that when a lower weight is assigned to the compact

decomposition, the result becomes worse for Bunch. This is in agreement with the observation in the previous section that Bunch performs better at coarse-grained levels.

A similar phenomenon takes place for KE but in the case of Linux and Bunch. It is quite intriguing that a similar deviation was observed for this measure in the previous section as well. Further investigation is required to determine what feature of this metric is responsible for this behaviour with regard to the Linux decompositions.

The results for EdgeMoJo are again in the reverse order than expected for the reasons described above. At the same time, the EdgeSim results follow the expected order except that the result for ALL is better than that for L in the case of Mozilla and ACDC, the only deviation from the expected order we observed for ACDC.

Finally, it is quite interesting to note that through all these experiments, the results produced by ACDC were rated higher that those of Bunch in approximately 50% of the experiments. Perhaps not surprisingly, MoJo rates the ACDC results consistently better, while EdgeSim does so for the results of Bunch.

Our overall observation from these experiments is that the END framework has revealed many interesting properties of the clustering algorithms we experimented with as well as the comparison methods we used as plugins. We are hopeful that reverse engineers that are more familiar with the software system they are dealing with, as well as the clustering algorithm they are using, will be able to benefit even more from the use of this framework.

In the next section, we explain the motivation for developing a new nested decomposition evaluation measure.

4.3 UpMoJo Motivation

The main benefit of the END framework is that it allows users to utilize existing flat evaluation measures that they might be already familiar with. However, the END framework does not specify the function that will be used to convert the similarity vector to a number. While this makes the framework more flexible, it also increases the responsibility of the user. In practice, it is often difficult to differentiate between different weighting functions as the framework does not provide any guidelines as to which functions are best suited in which situations. This can result in different users getting different results from END while comparing the same nested decompositions.

Moreover, one of the properties of the END framework is that a misplaced object closer to the root of the containment tree results in a larger penalty than if the misplaced object were deeper in the hierarchy. Consider the four decompositions `A`, `B`, `C`, and `D` shown in Figures 4.9 and 4.8. The only difference between them is the position of object `c.h`. The END framework would determine that the difference between `A` and `B` is larger than that of `C` and `D` (the similarity vector in the first case is $(1, 1)$, while in the second it is $(0, 1)$).

Finally, the END framework does not penalize for some trivial hierarchical mistakes. For example, the two nested decompositions shown in Figure 4.10 are equal according to END.

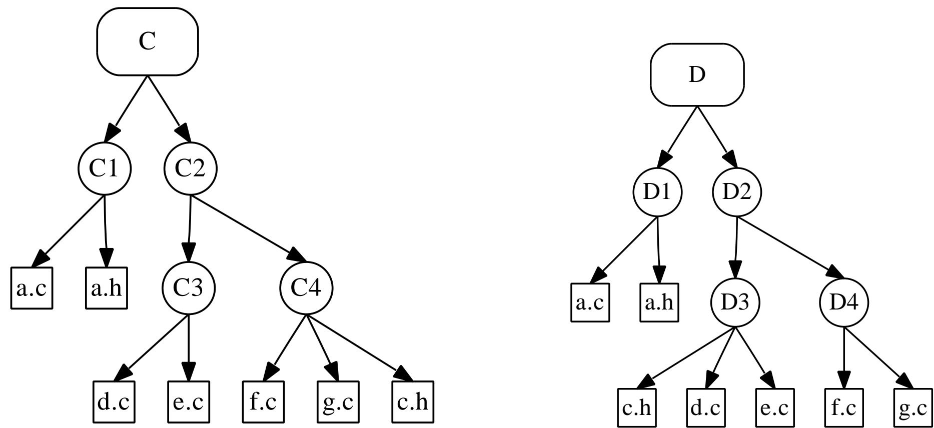


Figure 4.8: Decompositions `C` and `D`.

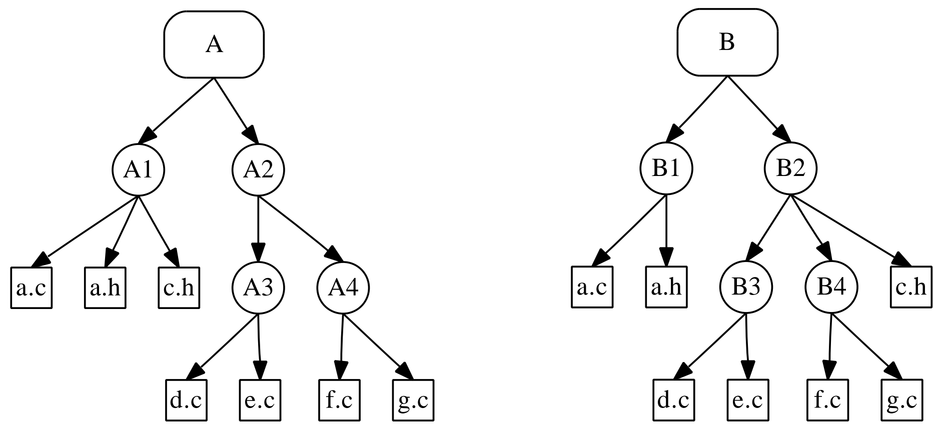


Figure 4.9: Decompositions `A` and `B`.

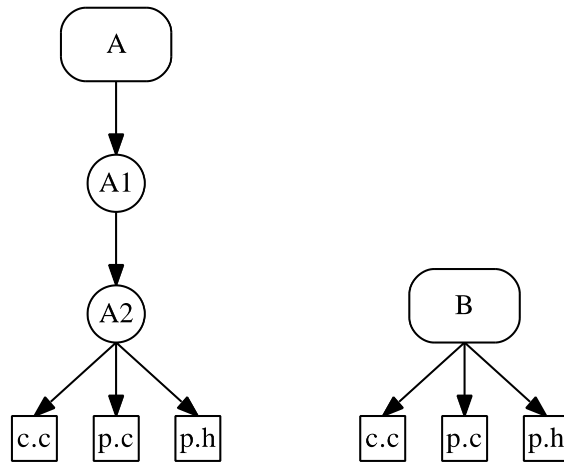


Figure 4.10: An example of a hierarchical mistake ignored by END.

The UpMoJo measure was developed to address these issues. It is described in detail in the following section.

4.4 UpMoJo distance

UpMoJo distance provides an evaluation method that is solely dependent on the two nested decompositions being compared. The details of UpMoJo distance calculation are explained in Section 4.4.1. Interesting properties of UpMoJo distance are discussed in Section 4.4.2. Evidence for the practical value of UpMoJo is presented in Section 4.4.3.

4.4.1 The UpMoJo Algorithm

This section presents the UpMoJo algorithm through the use of a running example. Let us assume that a software clustering algorithm has produced the nested decomposition

shown in Figure 4.11.

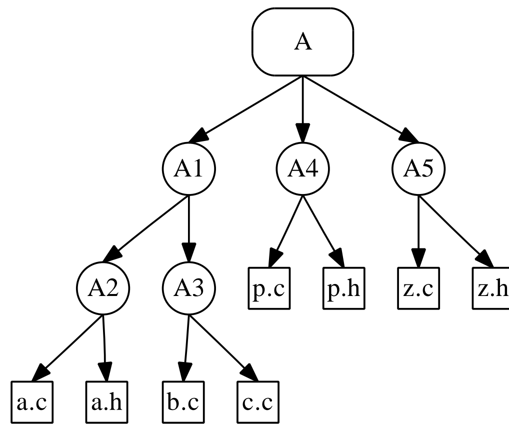


Figure 4.11: The containment tree of an automatic decomposition A.

In order to evaluate whether the clustering algorithm did a good job, we can compare decomposition A to one created by system experts. Let us assume that decomposition B shown in Figure 4.12 is the authoritative one for our example system.

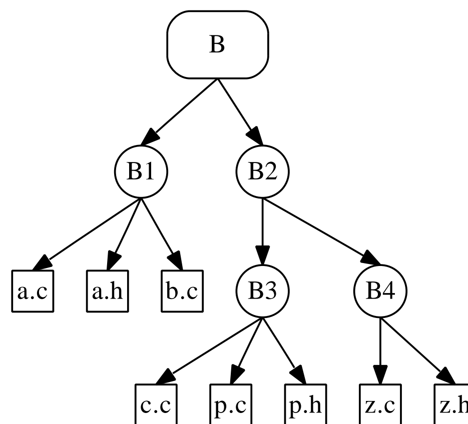


Figure 4.12: The containment tree of the authoritative decomposition B.

The UpMoJo distance between the two nested decompositions is defined as the number of operations one needs to perform in order to transform the containment tree of

decomposition A to the containment tree of decomposition B. The more operations this transformation requires, the more different the two decompositions are. The rest of this section describes how these operations are counted by our algorithm.

In the following, the *level* of a node n in the tree is again defined as the distance from the root to node n, e.g. subsystem A2 in decomposition A is at level 2. The root of a containment tree is at level 0.

We also refer to the system elements being clustered, i.e. the eight files in our example, as *entities*. As a result, a containment tree contains subsystems and entities. All leaf nodes in the tree are entities. All non-leaf nodes are subsystems.

The algorithm that calculates UpMoJo distance is as follows:

Find the set S of entities in level 1 in the authoritative nested decomposition. For each element o of S that is at a higher level than 1 in the automatic decomposition, transform the automatic decomposition by moving o to level 1. For each level that o has to move, increase the UpMoJo distance by 1. Objects that follow the same path move together, i.e. the UpMoJo distance is increased only for one of them. Each move of a set of entities to a lower level is called an Up operation and is explained in more detail in Section 4.4.2.

In our example, set S is empty, so no Up operations will take place. An example of the above process will be given for the algorithm's next iteration.

Next, we flatten the two decompositions to level 1, i.e. remove subsystems in levels other than 1 (this process does not modify the decompositions but rather creates copies of them). The two flat decompositions in our example are shown in Figures 4.13 and 4.14.

We now employ the MoJo distance measure for flat decompositions [TH99]. The MoJo

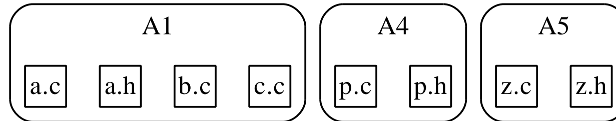


Figure 4.13: The flat decomposition of nested decomposition A.

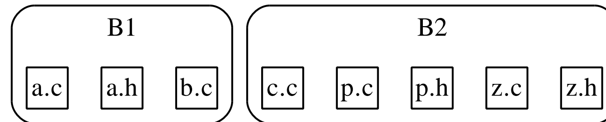


Figure 4.14: The flat decomposition of nested decomposition B.

distance between two flat decompositions of the same set of entities is the minimum number of Move and Join operations one needs to perform in order to transform one flat decomposition into the other. The two types of operations are defined as follows:

- **Mo**ve: Remove an entity from a subsystem and put it in a different subsystem. This includes removing an entity from a subsystem and putting it in a new subsystem by itself.

- **Jo**in: Merge two subsystems into one.

In our example, we can transform flat decomposition A to flat decomposition B by joining subsystems A4 and A5 into a new subsystem called A45 and moving c.c to A45 as well.

Having utilized MoJo to determine what are the necessary Move and Join operations for the flattened decompositions, we return to the original decomposition A and transform

it in the way suggested by the MoJo distance measure, i.e. join subsystems A4 and A5, and move entity c.c. This will result in the containment tree for decomposition A shown in Figure 4.15 (we denote it by A' to distinguish it from the original version). Subsystem A6 is created so that c.c is at the same level as before. The current total of the UpMoJo distance in our example is now 2 (1 Move and 1 Join operation).

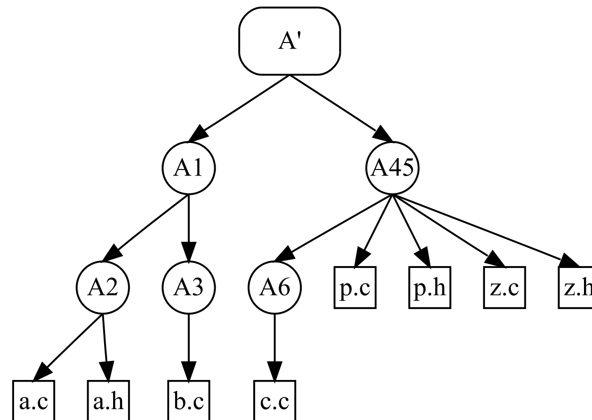


Figure 4.15: Decomposition A after the transformation indicated by MoJo.

Notice that decompositions A' and B contain the same number of top-level subsystems, and each of these subsystems transitively contain the same entities. This means that the process of transforming A to B can now continue recursively for each subsystem until A is transformed exactly into B.

The final value of the UpMoJo distance between decompositions A and B is the total number of Up, Move, and Join operations performed during the transformation process.

In our example, the next iteration will try to transform subsystem A1 into subsystem B1. Set S will contain entities a.c, a.h, and b.c (they are in level 1 with respect to subsystem B1). All three will need to move up, since they are initially in level 2 with respect

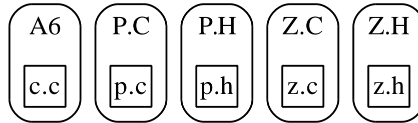


Figure 4.16: The flat decomposition of subsystem A45.

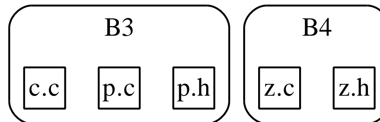


Figure 4.17: The flat decomposition of subsystem B2.

to subsystem A1. However, since two of them reside initially in the same subsystem, only two Up operations are required. This will increase the current total for the UpMoJo distance to 4. It should be easy to see that the flat decompositions will be exactly the same, so nothing further will be required for this subtree.

Finally, for subsystem B2, set S will be empty. The flat decompositions are shown in Figures 4.16 and 4.17. Entities directly under the root of the tree are considered to be in a separate subsystem of cardinality 1 in the flat decomposition.

Three further Move operations are required to transform the flat decomposition of A45 to the flat decomposition of B2. Entities p.c, p.h and z.h need to be moved. By performing these operations on the containment tree of decomposition A', we arrive to the containment tree shown in Figure 4.18, which is identical to the one of decomposition B (the names of the subsystems are immaterial).

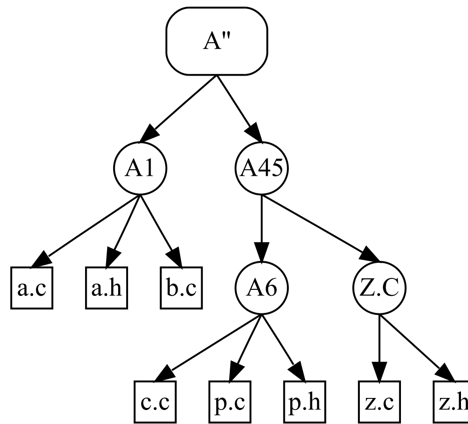


Figure 4.18: The final containment tree for decomposition A.

As a result, the final value of UpMoJo distance will be 7.

Pseudocode for the UpMoJo algorithm is shown in Figure 4.19.

4.4.2 UpMoJo Discussion

There are three properties of the UpMoJo algorithm that warrant further discussion:

1. The way the Up operation works.
2. The apparent lack of a Down operation.
3. The fact that UpMoJo does not attempt to compute the minimum number of Up, Move, and Join operations.

The Up operation is necessary when two different decompositions of the same software system have placed a given entity at a different hierarchical level. The definition of an Up operation is as follows:

```
UpMoJo(Test:Decomposition, Authoritative:Decomposition)

        Total:=0

        TRoot:=GetRoot(Test)

        if TRoot is leaf then return Total

        ARoot:=GetRoot(Authoritative)

        S := all entities from level 1 of

                 the authoritative nested decomposition

        UpOperations:=ExecuteUpOperations(S,Test)

        TestFlatten:=Flatten(Test)

        AuthoritativeFlatten:=Flatten(Authoritative)

        MoJoOperations:=ExecuteMoJoTransformation(Test,TestFlatten,

                           Authoritative,AuthoritativeFlatten)

        Total:= MoJoOperations + UpOperations

        For each object c from level 1 of Authoritatve

                T1 := GetSubDecomposition (Test, c)

                A1 := GetSubDecomposition(Authoritative,c)

                Total := Total + UpMoJo(T1, A1)

        end

        return Total

end
```

Figure 4.19: Pseudocode for the UpMoJo algorithm.

Up: Move an entity or a set of entities that initially reside in the same subsystem S to the subsystem that directly contains S.

The intriguing property of the Up operation is that one is allowed to move a set of entities with one operation. Figure 4.20 shows an example of such an operation. The intuition behind this lies with the fact that these entities have already been placed together by the software clustering algorithm which implies that they are related. The only problem is that they are not at the same hierarchical level as in the authoritative decomposition. It seems unfair that the algorithm be penalized for each entity individually, since the most important property, the fact that these entities are related, was discovered.

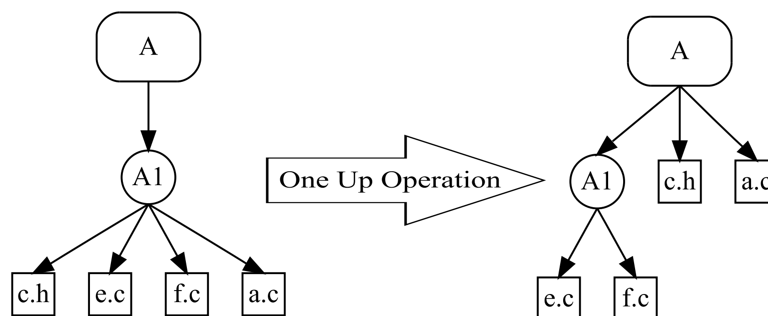


Figure 4.20: Example of 1 Up operation.

The second interesting property of UpMoJo is that there is no Down operation. At first sight, this seems to be strange since the possibility exists that an entity will be placed higher in the automatic decomposition than in the authoritative one. However, a downward movement is accomplished by the Move and Join operations. Introducing a Down operation would penalize twice for the same discrepancy between the two decompositions. The following example indicates how this works:

Consider the two decompositions shown in Figure 4.21. Suppose that we are transforming A to B. It would appear that a Down operation is required. The UpMoJo algorithm will accomplish the same effect implicitly as shown below.

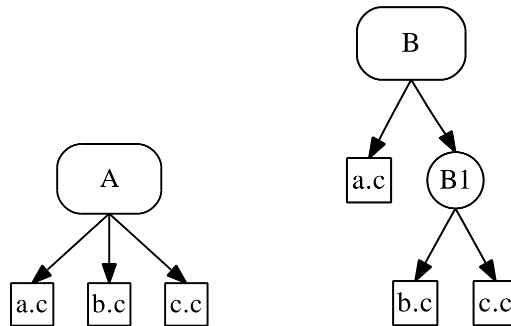


Figure 4.21: An example of an implicit Down operation.

The first step of UpMoJo is to perform any necessary Up operations. It is easy to see that no such operations are required in this example. Next, the flat decompositions for level 1 are calculated. These are shown in Figure 4.22 (as explained earlier, entities directly under the root of the tree, `a.c`, `b.c`, and `c.c` in this case, are considered to reside in separate subsystems of cardinality 1 in the flat decomposition). MoJo would indicate one Join operation. This will mean that entities `b.c` and `c.c` are now one level lower in the hierarchy.

(a) The flat decomposition of A (b) The flat decomposition of B

Figure 4.22: The flat decompositions of the decompositions in Figure 4.21.

Finally, the UpMoJo algorithm does not compute the minimum number of Up, Move and Join operations required to transform one nested decomposition to another. The rationale behind this is that minimization adds an exogenous criterion that may lead the algorithm to deviate from measuring the intended difference between decompositions. As a result, minimization may cause real differences between decompositions to be masked. To see how this can happen, let us define a new metric called MinUpMoJo that does compute the minimum number of operations needed to transform the containment tree of decomposition A to the containment tree of decomposition B.

In order to indicate why UpMoJo is a more appropriate measure, we will use the nested decompositions in Figure 4.23. Table 4.5 presents results from applying both UpMoJo and MinUpMoJo to these decompositions. MinUpMoJo has determined that both A and B are equally different from C. This result is misleading since B should be closer to C than A (the majority of the clusters in decomposition B have corresponding clusters in decomposition C, while no cluster from decomposition A has a corresponding cluster in decomposition C). The optimization behaviour of MinUpMoJo has masked the differences between A and C. On the other hand, the UpMoJo method ranked the nested decompo-

sitions correctly and confirmed that `B` is closer to `C`. Since MinUpMoJo may mask differences between two nested decompositions when the UpMoJo distance evaluates those decompositions correctly, MinUpMoJo distance is less suitable than UpMoJo distance.

	(A,C)	(B,C)
MinUpMoJo	4	4
UpMoJo	7	4

Table 4.5: MinUpMoJo and UpMoJo results.

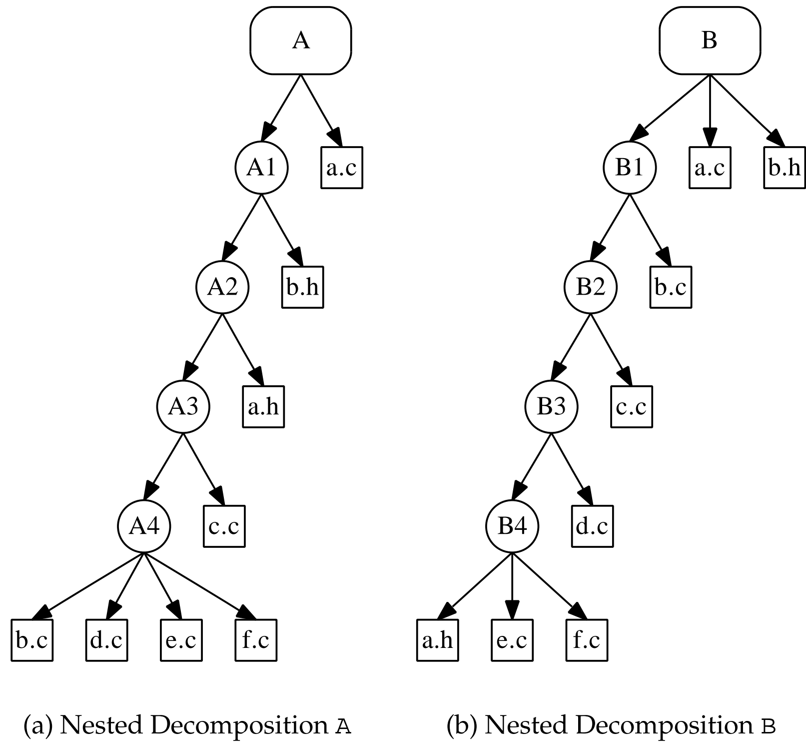
4.4.3 Experiments

The advantages of comparing nested software decompositions in a lossless fashion, i.e. without converting them to flat ones first, should be apparent by now. When one flattens a decomposition, important information that could distinguish two decompositions may be lost. While this is certainly true in theory, it is not apparent whether it makes a significant difference in practice.

The experiments presented in this section attempt to answer the following questions:

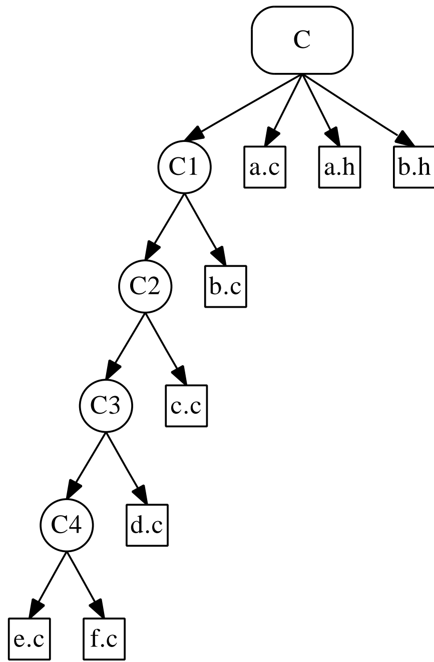
1. Does UpMoJo produce different results in practice than a comparison method for flat decompositions such as MoJo?

2. What are the practical differences between UpMoJo and the END framework using a trivial weighting function, such as equal weight for all values in the similarity vector?



(a) Nested Decomposition A

(b) Nested Decomposition B



(c) Nested Decomposition C

Figure 4.23: Limitation of MinUpMoJo distance.

To answer these questions, we utilize the congruity measure, that was presented in Section 3.4.3. This measure calculates correlation between two evaluation measures. In order to calculate the congruity measure we generated 100 simulated nested decompositions containing 346 entities with an average height of 4. The calculation of the congruity measure was repeated 10 times to ensure that our results are not based on an unlikely set of decompositions.

The values of the congruity measure when comparing MoJo and UpMoJo ranged from 35 to 44. This clearly indicated that there are significant differences between MoJo and UpMoJo. Using MoJo to compare nested decompositions runs the risk of producing results that do not reflect the inherent differences between the decompositions.

We calculated in the same way the congruity measure between the END framework and UpMoJo. In the case of END, we used MoJo as the plugin comparison method, and applied a weighting vector that contained equal weights for all values of the similarity vector. The same experiment setup as before was used (100 simulated decompositions, 10 repeats). The values of the congruity metric ranged from 25 to 42, indicating again a significant difference between END and UpMoJo. Section 4.3 mentions the main differences between END and UpMoJo. It is up to the reverse engineer to decide which set of features is more appropriate for their project.

Table 4.6 presents further experimental results that confirm the above findings (they also add the expected result that END is significantly different from MoJo). The table presents values for the congruity metric for an experiment setup of 100 simulated nested decompositions and various combinations of values for two generation parameters: D is

Comparison methods	D = 6 C = 4	D = 8 C= 4	D = 8 C = 20	D = 8 C = 60	D = 12 C = 60
MoJo and UpMoJo	37	42	50	43	46
MoJo and END	25	31	42	46	41
END and UpMoJo	32	41	36	41	50

Table 4.6: Congruity metric values for several experimental setups.

the maximum possible depth of the generated nested decomposition, and C is the maximum possible children for any subsystem in the generated decomposition.

These experimental results indicate clearly that the three comparison methods provide significantly different results.

This chapter presented two distinct approaches for the evaluation of nested decompositions of large software systems without having to lose information by transforming them to flat ones first. These are the only methods in the software clustering literature that address this problem.

The difference between the END framework and UpMoJo distance were presented in detail, so that a reverse engineer can choose which evaluation method suits their purposes better.

In the next chapter, we introduce a novel set of indicators that evaluate structural discrepancies between software decompositions.

5 Structure Indicators

Several software clustering evaluation methods have been developed [KE00, MM01a, ST04, ST07, TH99] to measure the similarity between an automatically created decomposition and a pre-existing authoritative one. However, while the existing methods have been quite useful, there are certain issues that can be identified.

Firstly, the existing measures often disagree in their evaluation. Despite having the same goal, that of quantifying how similar an automatic and an authoritative decomposition are, they frequently disagree in their ranking of different automatic decompositions for the same software system. The reasons for these discrepancies are not understood very well so far.

Secondly, existing measures provide a single number that is representative of the quality of the automatic decomposition. However, there could be many underlying reasons that result in higher or lower quality, since there are several ways in which an automatic decomposition can differ from an authoritative one. A better understanding of these partial differences would be beneficial to software maintenance activities.

We introduce a set of structure indicators that a software maintainer could use in parallel with existing measures in order to better understand the properties of automatically

created decompositions.

The structure of the rest of this chapter is as follows. Motivation for our work in evaluating structural differences between decompositions is presented in Section 5.1. Section 5.2 introduces the structure indicators that quantify these differences. We utilize these indicators in several experiments that demonstrate their usefulness in Section 5.3.

5.1 Structure Evaluation

While clustering evaluation methods, such as MoJoFM and KE, can be quite useful, certain aspects of these measures could be improved upon. In particular, we identify the following two issues:

1. The stated goal of both measures is the same: to evaluate the quality of an automatically created decomposition by comparing it to an authoritative one. However, it has been shown [AATW07, ST07] that the two measures often disagree in their ranking of different automatic decompositions for the same software system.

2. Both measures provide a single number that is representative of the quality of the automatic decomposition. However, as outlined below, there are several ways an automatic decomposition can differ from an authoritative one. Measures of these partial differences could be very useful for clustering evaluation.

The goal of the chapter is to investigate possible reasons for the discrepancies between MoJoFM and KE and to identify further metrics that a software maintainer could use in parallel with MoJoFM and KE. We begin by presenting a motivating example.

Let us consider a software system that contains 8 entities, such as classes or source files. Expert analysis has indicated that there are three subsystems. Figure 5.1 shows the authoritative decomposition `A` for this example system.

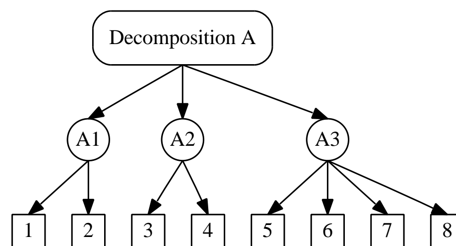


Figure 5.1: Authoritative decomposition `A`.

Figures 5.2 and 5.3 show two automatic decompositions of the same software system. Let us assume that software clustering algorithm SCA_B created decomposition `B`, while algorithm SCA_C created decomposition `C`.

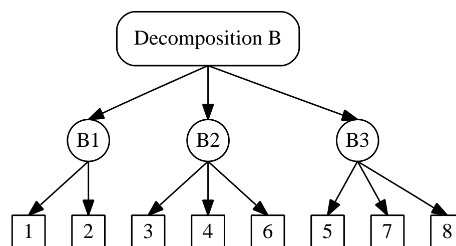


Figure 5.2: Automatic decomposition `B`.

While both algorithms have recovered the system structure fairly well, it should be clear that algorithm SCA_B should be more helpful to someone attempting to understand the high level structure of this software system. It has recovered all subsystems correctly with only minor discrepancies with regard to their contents. On the other hand, the fact that subsystems `C3` and `C4` are closely related was missed by algorithm SCA_C.

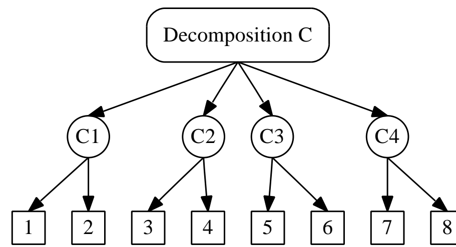


Figure 5.3: Automatic decomposition C.

Unfortunately, MoJoFM and KE do not agree with this assessment. Only one Move operation is required to transform B to A, while only one Join operation is required to transform C to A. As a result, MoJoFM considers B and C to be of equal quality. On the other hand, since KE does not penalize for joining clusters, its value for C will be 1, the maximum possible, while its value for B turns out to be 0.81. As a result, KE considers C to be the better automatic decomposition, contrary to our earlier assessment.

One can amplify the presented example by replacing every entity other than entity 6 with a large number of entities without affecting the validity of the example (the only change will be that the value of KE for B will go up, but it will still be less than 1).

This example illustrates that measures such as MoJoFM or KE are not sufficient for an accurate evaluation of a software decomposition. In Section 5.2, we present additional measures that augment MoJoFM and KE and capture the fact that decomposition B is closer to the authoritative one.

Furthermore, the above example presents a first hint at what might be the reasons for the occasional disagreements between MoJoFM and KE. The following example should help illuminate the situation further.

Let us consider a different software system that contains 16 entities. Its authoritative decomposition is shown in Figure 5.4.

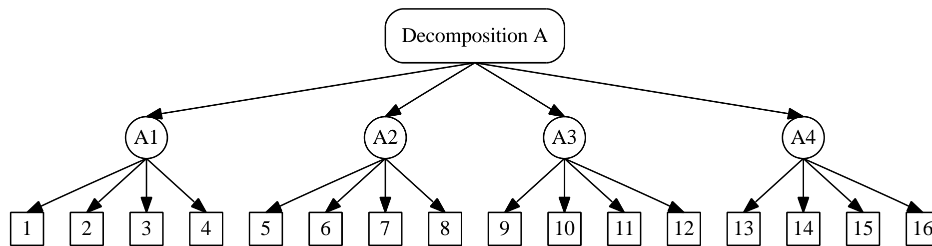


Figure 5.4: Authoritative decomposition `A`.

Figures 5.5 and 5.6 show two automatic decompositions of the same software system.

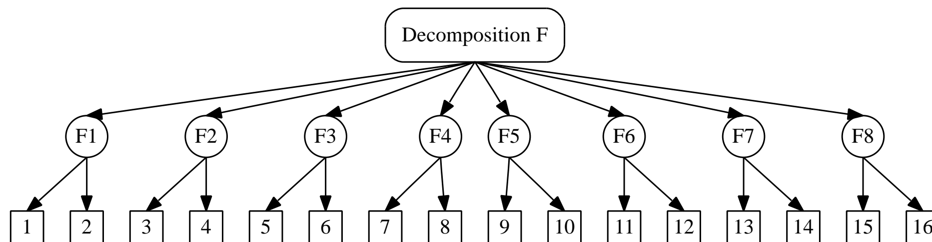


Figure 5.5: Automatic decomposition `F`.

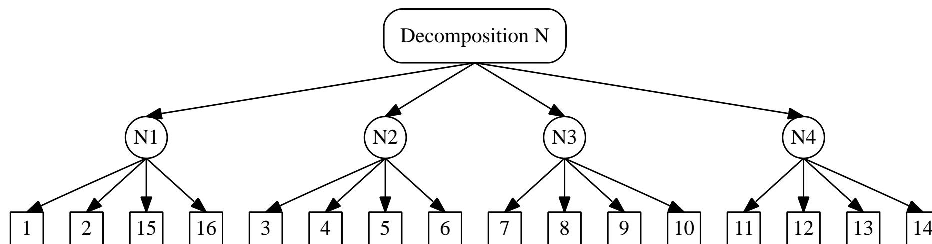


Figure 5.6: Automatic decomposition `N`.

While both automatic decompositions differ significantly from the authoritative one, they do so in a distinctly different fashion.

Method	Quality of F	Quality of N
MoJoFM	0.67	0.33
KE	1	0

Table 5.1: MoJoFM and KE values for decompositions F and N.

Decomposition F contains clusters whose elements are correctly related according to the authoritative decomposition. However, the close relation between many of the clusters is lost. We say that such a decomposition exhibits a large amount of *fragmentation*.

Decomposition N has the same number of clusters as the authoritative decomposition. However, many of the elements in these clusters are not correctly placed together. We say that such a decomposition exhibits a large amount of *noise*.

Table 5.1 presents the values of MoJoFM and KE for this example.

It should be evident by this admittedly extreme example that large amounts of fragmentation result in significantly higher values for KE, while large amounts of noise result in significantly lower values. The effect of such differences between decompositions could explain the occasional disagreements between MoJoFM and KE.

The two examples presented above provide the motivation for the work presented in this chapter. There is a need to evaluate the structure of an automatic software decomposition in isolation, i.e. without considering the effect of noise in the clusters. This would allow for accurate evaluation of the decompositions in the first example, and would enable us to investigate the differences between MoJoFM and KE in a controlled setting, e.g. between decompositions with similar structure. This investigation is discussed in Section

6.3.

The next section presents three distinct structure indicators we have developed. In Section 5.3 we utilize these indicators in a number of experiments.

5.2 Structure Indicators

This section introduces three indicators that can be used to evaluate the structure of an automatic decomposition. They are all based on the intuition that two decompositions have similar structure when a software engineer gets the same picture about the software system by reading either of these decompositions. The indicators are presented in the context of flat decompositions. If necessary, one can use the END framework to apply any of these indicators to a nested decomposition as well.

We will present the structure indicators through the means of an example that illustrates the need for all three of them. Figure 5.7 presents the authoritative decomposition A of a software system with 13 entities. Figure 5.8 presents an automatic decomposition T that will refer to as the "test" decomposition.

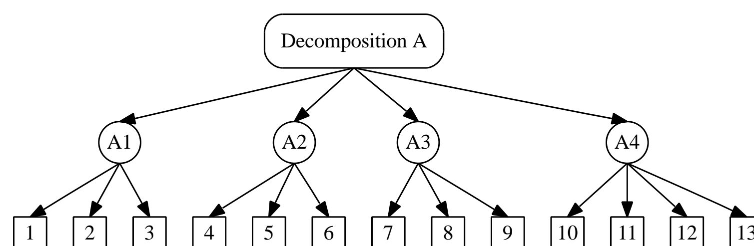


Figure 5.7: Decomposition A of a software system.

The following observations can be made about the two decompositions:

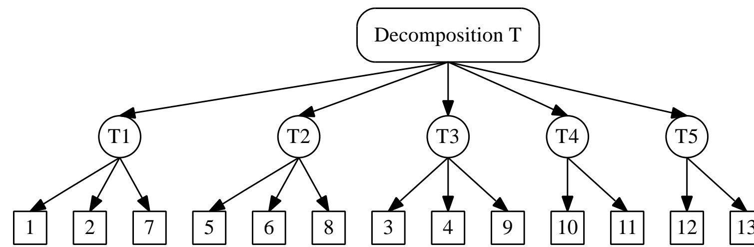


Figure 5.8: Decomposition T of the software system.

- Clusters A1 and A2 from the authoritative decomposition are similar to the T1 and T2 clusters from the test decomposition (a small amount of noise does exist however).

- Cluster T3 does not resemble any cluster in A. In other words, the clustering algorithm has identified a meaningless cluster.

- Cluster A3 does not resemble any cluster in T. In other words, information about cluster A3 is lost because it was never recovered by the software clustering algorithm.

- Cluster A4 is divided into two clusters T4 and T5 in the test decomposition.

Based on this example, we can identify three ways in which the structure of the test decomposition can differ from that of the authoritative one:

1. A cluster in the test decomposition does not have a corresponding cluster in the authoritative decomposition.

2. A cluster in the authoritative decomposition does not have a corresponding cluster in the test decomposition.

3. A cluster in the authoritative decomposition is fragmented into several clusters in the test decomposition.

The structure indicators presented in this section will gauge the magnitude of each type of difference presented above. In order to precisely define our indicators, we define the following:

Definition 5.2.1. A cluster T_i from the test decomposition is called a *segment* of cluster A_j from the authoritative decomposition if $\frac{|T_i \cap A_j|}{|T_i|} > 0.5$.

Cluster T_i is a segment of cluster A_j when the majority of the entities of T_i belong to A_j in the authoritative decomposition. This allows to define the three structure indicators as follows:

Extraneous Cluster Indicator (E): The number of clusters in the test decomposition that are not segments of any authoritative cluster. The value of E in our example is 1 due to cluster `T3`.

Lost Information Indicator (L): The number of clusters in the authoritative decomposition that do not have any segments in the test decomposition. In our example, L = 1 due to cluster `A3`.

Fragmentation Indicator (F): Let us denote by S the number of clusters in the test decomposition that are segments. The fragmentation indicator is defined as:

$$\begin{cases} \frac{S}{|A|-L} & \text{when} \quad |A| > L \\ 0 & \text{when} \quad |A| = L \end{cases}$$

It is a measure of the average number of segments for all authoritative clusters that do have at least one segment. In our example, the value of F will be $\frac{4}{4-1} = 1.33$.

We refer to the values of the three indicators as the ELF vector. In our example, the ELF vector would be (1,1,1.33).

Since the value of F can correspond to a number of different distributions for the number of segments per cluster, we can augment our three indicators with a *fragmentation distribution* graph.

Figure 5.9 shows the fragmentation distribution graph for our example. Each point in the x-axis represents a cluster in the authoritative decomposition that has at least one segment in decomposition T. The y-axis shows the number of segments for that cluster. Clusters are sorted in descending order of number of segments. More interesting fragmentation distribution graphs can be found in Section 5.3.

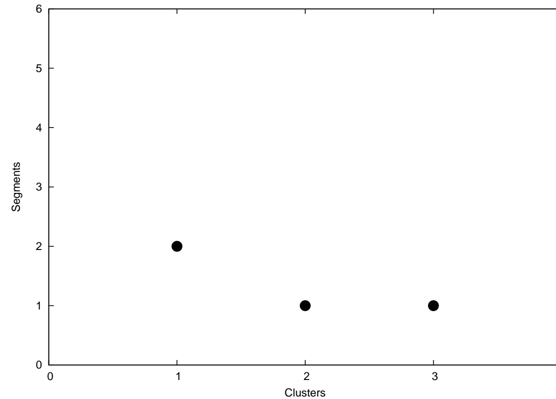


Figure 5.9: The fragmentation distribution graph for our example.

The three structure indicators presented above allow us to quantify the notion of similarity in decomposition structure. Test decompositions have similar structure to the au-

thoritative one when the ELF vector is (0,0,1). As these values start to increase, the structures of the two decompositions are becoming more and more dissimilar.

The three distinct indicators also enable us to investigate particular types of structural differences and how they affect the values of measures, such as MoJoFM and KE. For example, one can limit their investigation to decompositions with low E and L, and high F to determine how fragmentation affects the behaviour of these measures.

Finally, it is interesting to note that the structure indicators would evaluate the two decompositions in the first example of Section 5.1 more accurately. The better decomposition has indeed an ELF vector of (0,0,1), while the other decomposition has (0,0,1.33).

In the next section, we assess the usefulness of these indicators by utilizing them in a number of experiments with real systems.

5.3 Experiments

The goal of the experiments presented in this section is to demonstrate the usefulness of the structure indicators and to investigate the effect of different thresholds in structure evaluation. In the first series of experiments, we compare results obtained from existing comparison methods to results obtained by structure indicators. The results clearly show that the structure indicators provide additional evaluation information. In the second series of experiments, we compare results obtained using different thresholds to show that the choice of threshold does not affect the evaluation results significantly.

The experiments presented in this section were run on decompositions of two large

open source software systems, Linux and Mozilla. Apart from the authoritative decompositions of these two systems, we also created automatic decompositions by clustering them with two well-known software clustering algorithms, ACDC [TH00a] and Bunch [MMCG99].

Both the authoritative and the automatic decompositions for Linux and Mozilla are nested, i.e. subsystems can be subdivided into smaller subsystems which themselves may be subdivided etc. As a result, we were able to run experiments for both the flat[5] and the nested versions of these decompositions.

Table 5.2 summarizes the decompositions used in the following experiments. It also presents the abbreviations we will use for them in this section, as well as the number of clusters in each one. The height of each nested decomposition is also reported.

5.3.1 ELF evaluation

The target of the first series of experiments is to compare the results obtained from existing comparison methods to the results obtained by the structure indicators. We calculated the similarity between different flat decompositions using MoJoFM, KE, and the structure indicators. Table 5.3 presents a summary of the obtained results. The calculation of these values was practically instantaneous for all experiments.

These results allow for some interesting observations. First, the value of KE is always significantly lower than that of MoJoFM, a behaviour that has been observed in other similar experiments. It seems unlikely that an established tool such as Bunch has a quality

[5] We transformed the nested decompositions into flat compact ones (See Section 4.1).

Abbreviation	Description	Clusters	Height
LF	Linux Flat Authoritative Decomposition	7	-
LFA	Linux Flat ACDC Decomposition	130	-
LFB	Linux Flat Bunch Decomposition	45	-
MF	Mozilla Flat Authoritative Decomposition	10	-
MFA	Mozilla Flat ACDC Decomposition	461	-
MFB	Mozilla Flat Bunch Decomposition	50	-
LN	Linux Nested Authoritative Decomposition	123	6
LNA	Linux Nested ACDC Decomposition	163	3
LNB	Linux Nested Bunch Decomposition	478	4
MN	Mozilla Nested Authoritative Decomposition	268	5
MNA	Mozilla Nested ACDC Decomposition	596	4
MNB	Mozilla Nested Bunch Decomposition	1646	5

Table 5.2: Decompositions used in the experiments.

Method	LF - LFA	LF - LFB	MF - MFA	MF - MFB
MoJoFM	0.64	0.70	0.60	0.62
KE	0.33	0.18	0.49	0.22
E	8 (130)	7 (45)	37 (461)	20 (50)
L	2 (7)	3 (7)	0 (10)	1 (10)
F	24.9	9.5	42.4	3.3

Table 5.3: Comparison results for flat decompositions. For E and L, the total number of clusters is shown in parentheses.

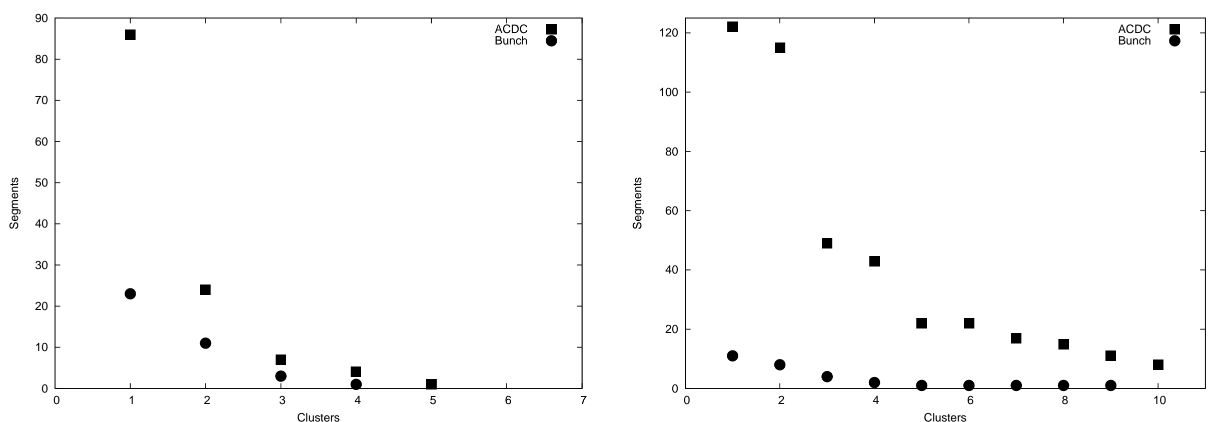
of 0.18 for Linux. A possible explanation is that the threshold that KE uses to decide whether two clusters match each other is too strict. Unfortunately, the implementation that was available to us did not allow us to change the fixed threshold value of 0.7.

A second interesting observation is that KE clearly considers that ACDC produces better results, while MoJoFM gives the edge to Bunch, albeit by a smaller margin. This rather large discrepancy can be explained somewhat when one compares the KE values to those of the fragmentation indicator F. There appears to be a significant correlation between these measures. Large amounts of fragmentation result in large KE values, something that was suggested by our extreme example in Section 5.1. Such amounts of fragmentation are, however, clearly counter-productive to a software maintainer. This would suggest that KE is not an appropriate evaluation method for algorithms that result in fragmented decompositions, which ACDC and Bunch seem to be.

To a lesser extent, a similar correlation can be detected between MoJoFM and the lost

information indicator L. While both algorithms perform well with regard to this indicator for Mozilla, there is a significant number of Linux clusters that was never recovered. However, MoJoFM indicates that both algorithms did better for Linux than Mozilla. This can probably be attributed to the fact that MoJoFM does not directly penalize for lost clusters, it simply reconstructs them with Move and Join operations. KE penalizes directly for such mismatches, which explains why the KE values are larger for Mozilla.

Finally, it is interesting to note that the results for the extraneous cluster indicator E are mostly positive, with the exception of the case of Mozilla and Bunch. This indicates that both algorithms produce meaningful clusters. The main problem for both algorithms appears to be large amounts of fragmentation, more so in the case of ACDC.



(a) Fragmentation Distribution Graphs for Linux

(b) Fragmentation Distribution Graphs for Mozilla

Figure 5.10: Fragmentation Distribution Graphs.

This conclusion is also supported by the fragmentation distribution graphs presented

in Figure 5.10. Both graphs show that only a small number of clusters have only one segment. Most authoritative clusters have several segments in the test decomposition. In the case of ACDC and Mozilla, the least fragmented cluster has 8 segments. A software maintainer will have a significantly different view about the software system by studying these test decompositions. This finding indicates that an important research direction for both algorithms, especially for ACDC, is to be able to produce less fragmented clusters.

We also performed a similar set of experiments using the nested version of all decompositions to investigate whether the above observations hold when all levels of a hierarchical decomposition are considered, as opposed to only the first one. Since no KE plugin for the END framework exists at this time, the experiments were performed only for MoJoFM and the structure indicators. These results are presented in Table 5.4.

Method	LN - LNA	LN - LNB	MN - MNA	MN - MNB
END (MoJoFM)	47.45	53.06	49.25	53.64
E	27.08 (163)	106.82 (478)	116.89 (596)	188.71 (1646)
L	33.68 (123)	29.87 (123)	65.05 (268)	72.3 (268)
F	11.21	4.79	19.55	3.21

Table 5.4: Comparison results for nested decompositions. For E and L, the total number of clusters is shown in parentheses.

The results are not as clear-cut as in the case of flat decompositions but they do not contradict any of the observations. It is interesting to note though that the good performance of the two algorithms with respect to the lost information indicator L is no longer

evident. This indicates that both algorithms perform better at a coarse level of granularity but are unable to recover several finer-grained clusters. However, it is clear, that the fragmentation problem remains the most important one even for nested decompositions.

5.3.2 Segment Threshold Evaluation

The definition of segment in Section 5.2 uses a threshold value of 0.5. In this set of experiments, we investigate the effect that different thresholds could have in our results, by substituting an alternative threshold. We chose the value of 0.7 which is the threshold used by KE in a similar situation. The set of experiments with flat decompositions presented in the previous section was repeated with the new threshold. The results are shown in Table 5.5.

Method	LF - LFA	LF - LFB	MF - MFA	MF - MFB
MoJoFM	0.64	0.70	0.60	0.62
KE	0.33	0.18	0.49	0.22
E	29 (130)	21 (45)	71 (461)	31 (50)
L	2 (7)	4 (7)	0 (10)	4 (10)
F	20.0	8.0	39.0	3.2

Table 5.5: Comparison results for flat decompositions using a segment threshold of 0.7.

The results show that indicator F decreases but still remains high, while indicators E and L increase in value compared to the values of the corresponding indicators calculated with threshold 0.5. This is additional evidence that supports our conclusion about the

significant structure differences between test and authoritative decompositions. More importantly, it shows that our choice of segment threshold does not affect the evaluative power of the structure indicators.

This chapter introduced a novel set of structure indicators that can augment the evaluation picture provided by traditional comparison methods, such as MoJoFM and KE. Several sets of experiments indicate that these structure indicators can be valuable in better understanding the properties of automatically created decompositions, while they can also help software clustering researchers investigate the similarities and differences of existing comparison methods.

In the next section, we address an open question related to software clustering, that of the comparability of software clustering algorithms. Also, we will investigate possible reasons for the discrepancies between two of the more popular evaluation methods, MoJoFM [WT04b] and KE [KE00].

6 Algorithm Comparability

Perhaps the most well-known tenet of clustering research is that there is no single correct answer to a clustering problem [Eve93]. There are many, equally valid ways to decompose a data set into clusters. Similarly, in the context of software clustering, there are many, equally valid ways to decompose a software system into meaningful subsystems. Different software clustering algorithms employ different clustering objectives producing different results. All such clustering results could be useful to someone attempting to understand the software system at hand.

It is, therefore, rather strange that the typical way to evaluate a software clustering algorithm is to compare its output to a single authoritative decomposition, i.e. a decomposition created manually by a system expert. If the authoritative decomposition was constructed with a different point of view in mind than the algorithm, the evaluation results will probably be biased against the algorithm. However, no better way to evaluate software clustering algorithms exists at the moment.

In this chapter, we introduce and quantify the notion of software clustering algorithm *comparability*. Comparable algorithms use the same or similar clustering objectives, produce similar results, and can therefore be evaluated against the same authoritative de-

composition. Non-comparable algorithms use different clustering objectives, produce significantly different results, and cannot be evaluated by direct comparison to an authoritative decomposition.

Being able to distinguish between comparable and non-comparable algorithms allows us to define families of algorithms that share common clustering objectives. Conventional evaluation techniques apply only within the context of a given algorithm family.

As expected, our experimental results clearly show that several existing software clustering algorithms belong in different families, and therefore should not be directly compared against each other.

To address this issue, we introduce a novel process for clustering evaluation called LimSim. It stipulates that comparison based on authoritative decompositions should only be done between comparable algorithms if possible. At the same time, it allows for more accurate evaluation since it employs multiple simulated authoritative decompositions.

Finally, in this chapter, we investigate possible reasons for the discrepancies between two of the more popular evaluation methods, MoJoFM [WT04b] and KE [KE00].

The structure of the rest of the chapter is as follows. The notion of comparability as well as the way we quantify it is presented in Section 6.1. Section 6.2 introduces LimSim and applies it to several existing algorithms. Finally, experiments that investigate the relation between MoJoFM and KE are presented in Section 6.3.

6.1 Clustering Algorithm Comparability

The various software clustering algorithms published in the literature have different clustering objectives, i.e. assumptions on what constitutes a meaningful decomposition of a software system. This leads to clustering results that, while significantly different from each other, offer equally valid points of view on the system's high level structure. Evaluating such clustering results by comparing them to a single authoritative decomposition, that may or may not have been created with the same clustering objective in mind, can lead to significantly biased results.

For this reason, we define that two clustering algorithms that produce significantly different results are *non-comparable*, i.e. they should not be compared against the same authoritative decomposition since the obtained results will be meaningless if not misleading. Conversely, we define two clustering algorithms to be *comparable* when they produce similar results. Comparable algorithms can be compared using an authoritative decomposition.

We quantify what is meant by "similar clustering results" in Section 6.1.1. Experiments to determine the comparability of existing software clustering algorithms are presented in Section 6.1.2.

6.1.1 Quantifying Comparability

In this section, we present a method that determines whether two software clustering algorithms A and B are comparable or not.

Our method begins by creating a large number of MDGs using the generation algorithm presented in Section 3.1. The two algorithms are applied to all MDGs. In order to measure the similarity between the various clustering results, we use the MoJoFM measure [WT04b]. A MoJoFM value of 100 means that the two algorithms construct the exact same decomposition. A MoJoFM value of 0 means that the two algorithms produce decompositions that are completely different, i.e. they disagree about the placement of all entities. Since MoJoFM is non-symmetric, we define the similarity between clustering algorithms A and B as $\min(MoJoFM(A, B), MoJoFM(B, A))$.

This allows us to compute the average similarity μ across all generated MDGs, as well as the standard deviation σ.

We consider two algorithms to be non-comparable if they produce significantly different results across a variety of software systems. More precisely, we define that two algorithms are non-comparable if in the majority of cases they produce results that are more different than similar, i.e. the similarity is less than 50%.

We use the well-known 3σ rule [Puk94] to capture the notion of "the majority of cases". The 3σ rule states that 99.73% of the values of a normal distribution lie within 3 standard deviations of the mean. This allows us to disregard outliers that occur very infrequently. As a result, our definition becomes:

Two software clustering algorithms are non-comparable if it holds that:

$$\mu + 3\sigma < 50$$

If the above does not hold, then the algorithms are considered comparable. This definition

may appear biased towards making algorithms look comparable, since one could have used only the average in the definition. However, as will be shown in the next section, even with this definition most major algorithms are shown to be non-comparable.

6.1.2 Experiments

The goal of the experiments presented in this section is to validate whether well-known software clustering algorithms are comparable. In addition, we present experiments whose purpose is to determine whether the similarity metric or the update rule function has the largest impact on the output decomposition. Finally, our experiments demonstrate the ability to deduce properties of a software clustering algorithm by studying the properties of comparable well-known algorithms.

Our experiments require a large number of module dependency graphs (MDGs) extracted from software systems. Since this is impractical to do, we generated the required number of factbases using the method explained in Section 3.1. For the experiments presented in this section, the following constraints were activated: No multiple edges, no loops, and low clustering coefficient.

The first set of experiments attempts to determine the comparability of the following clustering algorithms: Bunch [MMR+98], ACDC [TH00a], LIMBO [AT05], and several hierarchical clustering algorithms. We compared the performance of these algorithms on 10 simulated MDGs. The results of these experiments are presented in Table 6.1.

For our experiments, we used six different hierarchical clustering algorithms: all possible combinations of three update rule functions (complete linkage, weighted average,

Algorithm pair	μ	σ	$\mu + 3\sigma$	Comparable	Linux	Mozilla	TOBEY
Hierarchical and ACDC	35.07	6.74	57.29	Yes	41.6	26.92	28.12
Bunch and ACDC	25.07	7.66	48.35	No	31.32	19.97	34.06
Hierarchical and Bunch	16.47	4.77	30.78	No	33.34	20.56	19.03
LIMBO and ACDC	10.27	5.03	25.36	No	16.08	5.37	16.25
Bunch and LIMBO	8.28	3.33	18.27	No	9.09	5.35	15.77
LIMBO and Hierarchical	4.30	2.14	10.27	No	14.72	11.95	15.56

Table 6.1: Pair-wise comparability results for the selected clustering algorithms.

and unweighted average) and two similarity metrics (Jaccard and Sorensen–Dice). The cut-point heights were chosen so as to maximize the similarity values (again possibly biasing the results in favour of comparability). The designation Hierarchical in Table 6.1 refers to all six variations, i.e. the values reported are averaged across all hierarchical algorithms.

In order to determine how realistic the simulated MDGs are, Table 6.1 also provides the similarity values when the various algorithms are applied to three real systems: Linux, Mozilla, and TOBEY. With the exception of the LIMBO-Hierarchical pair, all real similarity values fall within one standard deviation of the average simulated similarity values. This is a strong indication that the simulated MDGs closely resemble graphs from real systems.

An immediate observation is that most of the selected algorithms are not comparable to each other. For instance, the decompositions created by LIMBO are significantly differ-

Algorithm pair	μ	σ	$\mu + 3\sigma$	Comparable
ACDC and Complete Linkage with Jaccard	38.63	6.85	59.18	Yes
ACDC and Complete Linkage with Sorensen–Dice	40.72	4.53	54.31	Yes
ACDC and Weighted Average with Jaccard	33.46	4.57	47.1	No
ACDC and Weighted Average with Sorensen–Dice	32.88	8.29	57.75	Yes
ACDC and Unweighted Average with Jaccard	31.67	4.71	45.8	No
ACDC and Unweighted Average with Sorensen–Dice	32.98	7.65	56.83	Yes

Table 6.2: Comparability of ACDC to Hierarchical Algorithms.

ent than those created by the other software clustering algorithms. A possible explanation is the fact that LIMBO has a unique objective when constructing software decompositions: the minimization of information loss.

The only case where a possible pair of comparable algorithms may be found is in the first row of Table 6.1. ACDC appears to be comparable to our set of hierarchical algorithms. In order to investigate further, we compared ACDC to each individual hierarchical algorithm, as shown in Table 6.2.

A first observation is that ACDC is comparable to a majority of the selected hierarchical algorithms. This can probably be attributed to ACDC's hierarchical nature: Subgraph dominator pattern instances are composed in agglomerative fashion in order to achieve the goal of limiting the number of elements in each subsystem [TH00a].

Moreover, the close comparability of ACDC to complete linkage suggests that ACDC constructs cohesive clusters, a well-known property of the complete linkage algorithm

Update Rule Function	μ	σ
Complete Linkage	99.98	0.05
Weighted Average	66.32	9.34
Unweighted Average	73.94	10.35

Table 6.3: Comparability results for constant update rule function and varying similarity metric (Jaccard and Sorensen–Dice).

[AL99, DB00]. This fact also supports our earlier finding that ACDC and Bunch are not comparable. Bunch attempts to produce decompositions with high cohesion and low coupling, while ACDC focuses mostly on high cohesion.

The above findings illustrate an important benefit of assessing algorithm comparability. One may be able to deduce properties of an algorithm they are considering for a software clustering task by studying the properties of comparable, well-known algorithms.

The next set of experiments involves only the 6 selected hierarchical algorithms, and its purpose is twofold:

1. Act as a control experiment. Since all hierarchical algorithms have the same objective, they should be comparable to each other.

2. Determine whether the similarity metric or the update rule function has the largest impact on the output decomposition.

Algorithm Pair	Similarity Function	μ	σ
Complete Linkage and Weighted Average	Jaccard	39.98	9.87
Complete Linkage and Unweighted Average	Jaccard	37.49	10.04
Weighted Average and Unweighted Average	Jaccard	49.75	7.34
Complete Linkage and Weighted Average	Sorensen–Dice	44.02	17.70
Complete Linkage and Unweighted Average	Sorensen–Dice	45.45	18.07
Weighted Average and Unweighted Average	Sorensen–Dice	54.24	14.21

Table 6.4: Comparability results for constant similarity metric and varying update rule function.

Table 6.3 presents experimental results when the update rule function is constant, while Table 6.4 presents results when the similarity function is constant.

A first observation is that all hierarchical algorithms are comparable to each other, as expected. As a result, they form the only known family of software clustering algorithms so far (it is debatable at this point whether ACDC should be considered a member of this family).

The results also suggest that the update rule has a larger impact on the output of a software clustering algorithm than the similarity function. In fact, in the case of complete linkage the similarity metric appears to make almost no difference, a very interesting result on its own.

Moreover, it is evident that the Jaccard similarity metric has a larger impact than the Sorensen–Dice metric (the comparability values are consistently smaller for Jaccard). This

corroborates findings that the Sorensen–Dice similarity metric works well in the software context [DB00, LZN04].

The experimental results presented above seem to validate the assumption we hinted at in the beginning of Section 6.1. Two clustering algorithms that have different objectives are by definition non-comparable, i.e. their results should not be compared against a single authoritative decomposition.

Since most of the clustering algorithms published in the literature have different objectives however, it is clear that a new process for clustering evaluation is necessary. In the next section, we introduce an approach that addresses this issue.

6.2 A new clustering evaluation process

The fact that several published software clustering algorithms are not comparable to each other allows us to see the way we currently evaluate clustering algorithms under a different light. We can identify two major issues that need to be addressed:

1. Evaluation is often performed by comparing a new algorithm with non-comparable existing algorithms using a single authoritative decomposition. As pointed out earlier, this can lead to biased results since the authoritative decomposition is only based on at most one of the clustering objectives used by the algorithms.

2. Only a small number of reference software systems have available authoritative decompositions. It is not feasible to assess the effectiveness of a software clustering

algorithm using only a handful of data points.

In this section, we present a novel process for the evaluation of software clustering algorithms, called LimSim, that addresses the aforementioned issues. The proposed method derives its name from the following two features:

1. It stipulates that evaluation of a clustering algorithm should be **lim**ited to comparable algorithms when possible. If a better performing comparable algorithm exists, then the algorithm being evaluated may need to be discarded or slated for improvement. If no comparable algorithms exist, comparison is still possible thanks to the following feature.

2. It allows evaluation based on a large number of **sim**ulated software systems with available authoritative decompositions. This feature is explained in detail in the following section.

6.2.1 Simulated Authoritative Decompositions

The number of large software systems with an available authoritative decomposition is rather small. The process introduced in this section allows for the creation of a large number of simulated software systems with available authoritative decompositions.

We start with a real software system, such as Linux or TOBEY, for which there is an MDG and an authoritative decomposition. Next, we apply a series of small modifications to the MDG with corresponding modifications for the authoritative decomposition. Each

modification is designed in such a way as to ensure that the resulting decomposition is indeed authoritative for the resulting MDG.

After a large number of randomly selected modifications has been applied, we will have an MDG that is significantly different than the original one for which an authoritative decomposition exists. By repeating this randomized process many times, we can obtain a large population of simulated software systems with available authoritative decompositions.

The effectiveness of this method clearly relies on a set of modifications that are designed to preserve the authoritativeness of the resulting decomposition. Our current implementation includes five such modifications presented below. The first modification is presented in detail, while only a high level description is provided for the remaining four.

Merge two modules: The intuition behind this modification is that if two modules that belong in the same cluster are combined into one, the resulting module must also belong in the same cluster since its dependencies to the rest of the system are a combination of the dependencies of its constituents.

This modification is applied using the following process:

1. Randomly select a cluster S that contains at least X entities, where X is a configurable parameter with a default value of 5. This restriction avoids merging all elements of a cluster, which would result in a meaningless cluster.

2. Randomly select two modules `A` and `B` from S.

3. Create a new module `C` in the MDG. All incoming and outgoing edges from/to

modules A and B are moved to module C. Modules A and B are deleted from the MDG.

4. Delete modules A and B from the authoritative decomposition and add module C in S.

Figure 6.1 shows the results of this modification on a sample MDG.

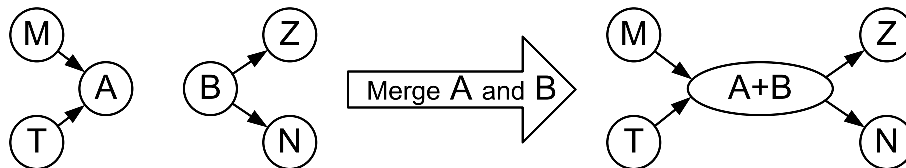


Figure 6.1: Example of merging modules A and B.

Split a module: This modification is the opposite of the Merge modification, and it can be viewed as the simulation of refactoring a complex module into two simpler ones. A randomly selected module loses some of the edges attached to it. These edges are assigned to a newly created module. The new module is assigned to the same cluster as the original module because the relation between the two modules and the rest of the system has not changed.

Copy a module: This modification creates a new module that has the same dependencies to the rest of the system as a randomly selected existing module. For this reason, the new module is assigned to the same cluster as the old one.

Add an edge: This modification adds an edge between two modules that belong to the same cluster. It applies only to modules that do not have any dependencies to other clusters, so the authoritative decomposition remains the same.

Remove cycle: This modification removes one edge in order to break circular dependencies. It only applies when all the modules involved in the circular dependency do not have any dependencies on modules in other clusters.

In the next section, we explain how the simulated software systems created using these modifications can be used to evaluate software clustering algorithms.

6.2.2 LimSim

The main goal of LimSim is to provide researchers with valuable information that will allow them to assess the effectiveness of a new software clustering algorithm. The best way to do so, is to compare the new algorithm with existing comparable algorithms. This can be done by running comparability experiments such as the ones in Section 6.1. This can lead to one of two outcomes:

1. Comparable algorithms are found. The properties of the new algorithm can then be estimated based on the properties of the most comparable existing algorithm. The upcoming comparison against simulated authoritative decompositions will be limited to comparable algorithms only.

2. No comparable algorithms are found. This shows that the new algorithm has a significantly different clustering objective from existing algorithms. The simulated authoritative decomposition comparison will be performed against non-comparable algorithms. While this is not ideal, most of the bias due to using a single authori-

tative decomposition is removed since the simulated decompositions can be quite different from the original one.

Next, LimSim evaluates the new software clustering algorithm on a large number of simulated software systems. For each real software system with an available authoritative decomposition, LimSim compares the results of the new algorithm against existing algorithms on a large number of simulated software systems created using the process described in the previous section.

For this purpose, a similarity metric such as MoJoFM [WT04b] or KE [KE00] can be used. The average value and standard deviation across all simulated systems can give a researcher a good indication of the effectiveness of the new algorithm. If comparable algorithms have been found, then the similarity values should demonstrate whether the new algorithm is an improvement or not.

We have developed a framework that implements LimSim. It automatically generates simulated software systems and their authoritative decompositions, evaluates clustering algorithms, and calculates statistic measurements. The framework can be extended by adding new similarity metrics, or other modifications for the process of creating simulated software systems.

Before presenting experiments applying LimSim to real software systems, it is interesting to point out two properties of LimSim that may not be immediately obvious:

1. An MDG may be created based on a set of dependencies that favours some algorithms but not others. By randomly changing the MDG, possible bias in its creation

may be removed. This allows focusing the evaluation process on the algorithm itself. This is the only evaluation method that does this.

2. By selecting a particular set of modifications one can examine the behaviour of an algorithm in a particular context. For example, the behaviour of an algorithm when applied on highly cohesive systems can be studied by allowing only modifications that increase cohesion, such as merging two modules or adding an edge.

In the next section, we present a set of experiments using LimSim.

6.2.3 LimSim Experiments

The goal of the following experiments is to demonstrate the usage of LimSim. Based on two real systems, Linux and TOBEY, 100 simulated software systems were created. These were clustered using 6 different algorithms and the results were compared to the simulated authoritative decompositions using MoJoFM. The results are presented in Table 6.5.

Some of the obtained results agree with previous studies. For instance, single linkage has the worst performance than all the algorithms, as shown before [MB04].

On the other hand, hierarchical algorithms have been shown to perform better than tailor-made software clustering algorithms, like ACDC and Bunch, only when the cut-point height is a posteriori selected to maximize their performance [AT05]. However, using LimSim, they perform better than ACDC in the case of Linux, and better than Bunch in the case of TOBEY (with the exception of single linkage) even when a pre-determined

Algorithm	Linux μ (σ)	TOBEY μ (σ)
ACDC	62.33 (3.01)	61.62 (1.56)
Bunch	74.42 (2.85)	56.32 (6.39)
Complete Linkage	64.25 (1.01)	56.57 (1.11)
Single Linkage	58.52 (0.96)	18.80 (0.65)
Weighted Average	64.66 (3.64)	57.27 (2.03)
Unweighted Average	66.27 (2.18)	57.99 (1.91)

Table 6.5: LimSim results for Linux and TOBEY using MoJoFM as similarity.

cut-point is used (we used a fixed cut-point of 0.3 for all experiments). This indicates that LimSim provides a different look at evaluation, and may be removing bias against the hierarchical algorithms that may have existed before.

In the next section, we present an experimental study that attempts to identify possible reasons for the discrepancies between two of the more popular evaluation methods, MoJoFM and KE.

6.3 MoJoFM and KE

The stated goal of both MoJoFM and KE is to quantify the quality of a test decomposition with respect to an authoritative one. However, they attempt to measure quality in different ways often resulting in contradictory results [AATW07, AT05]. The objective of

the experiments in this section is to investigate the relationship between the MoJoFM and KE measures when the decompositions being compared are limited to a a reduced search space. We define the following two spaces:

- **Structure reduction.** The test and authoritative decompositions have the same structure, i.e. the ELF vector of the test decomposition is (0,0,1).
- **Comparability reduction.** The two decompositions are produced by comparable software clustering algorithms.

In the following, we present experiments using decompositions from these spaces.

6.3.1 Structure reduction

In the experiments presented below we attempted to remove the effect that structure discrepancies between the test and authoritative decomposition may have in the measures' assessment. The structure indicators defined in Section 5.2 allowed us to compare MoJoFM and KE in a reduced search space, i.e. when the decompositions involved have the same structure. In other words, the ELF vector between the test and authoritative decompositions in this set of experiments was always (0,0,1). To evaluate the relationship between the two measures, the congruity metric (see Section 3.4.3) was calculated. The congruity metric is a value between 0 and 100. Values close to 0 or 100 indicate strong correlation, while values closer to 50 indicate lack of correlation.

The experiments were performed on a series of 100 randomly generated decompositions. The number of entities in these decompositions was between 1000 and 2000, while

the number of clusters varied between 10 and 20. The congruity metric value was always in the interval [42,49]. This indicates a lack of correlation between MoJoFM and KE even when they are comparing decompositions with the same structure. The randomized experiment was repeated 10 times with similar results.

A possible explanation for this rather surprising result (given that the goal of the two measures is the same) is the way they penalize for misplaced entities. MoJoFM applies the same penalty for any misplaced entity (one Move operation). KE applies a variable penalty depending on cluster size. When an entity is misplaced in a big cluster then the penalty is smaller than that for a misplaced entity in a small cluster. For example, misplacing a single entity in a decomposition of 2 clusters results in a KE value of 0.71 when the clusters have sizes 2 and 5. If the cluster sizes are 2 and 25, the KE value is 0.81.

It is also interesting to note that the value of KE was always smaller than that of MoJoFM in this set of experiments. For instance, the range of MoJoFM values was [0.97,1] while the corresponding range for KE was [0.83,1]. KE had values in the range [0.89, 0.98] when MoJoFM was held constant at 0.99.

The above results indicate strongly that MoJoFM and KE have significantly different behaviour even when they are applied to decompositions of similar structure.

6.3.2 Comparability reduction

The goal of the last set of experiments is to answer the following question: Is the behaviour of the two measures more congruent when evaluating comparable algorithms?

The experiments were performed on a series of 100 decompositions generated using

comparable algorithms. In the experiments, we used the complete linkage, single linkage, weighted average and unweighted average clustering algorithms. The number of entities in these decompositions was 1000.

The congruity metric value was always in the interval [43, 51]. This indicates a clear lack of correlation between MoJoFM and KE even when they are comparing decompositions produced by comparable algorithms.

The experimental evidence shows that the two measures have fundamental differences that cannot be reconciled. As a result, researchers need to select the comparison method whose properties better suit the needs of their task at hand.

This chapter introduced and quantified the notion of comparability for software clustering algorithms. Experiments showed that many of the published algorithms for software clustering are non-comparable.

A novel process for the evaluation of software clustering algorithms was also presented. It utilizes simulated software systems with authoritative decompositions in order to allow comparison across a large number of data points. Also, we presented experiments that showed the usefulness of our approach.

Finally, we determined that the MoJoFM and KE evaluation measures do not exhibit congruent behaviour even when used to evaluate decompositions that either have the same structure or are created by comparable algorithms. We provided a theoretical explanation of this surprising fact.

In the next chapter, we present methods for selecting and improving software clustering algorithms.

7 Methods for selecting and improving software clustering algorithms

Software clustering is a research area that has been developing for a long time. Many different software clustering algorithms have been developed [DYM+08, PHLR09, MM09] as presented in Chapter 2. Most existing algorithms have been applied to particular software systems with considerable success. This maturity of software clustering increases the importance of the following problems:

- Since software clustering algorithms are often heuristic, the quality of the produced decomposition may be dependent on whether the software system was a good match for the heuristic. Although the selection of a software clustering algorithm plays a key role for the construction of a meaningful decomposition, there are no guidelines on how to select an appropriate algorithm.
- The complexity of software clustering algorithms makes the understanding of their weaknesses and strengths difficult. Understanding a software clustering algorithm would be easier if all relevant information was presented in a standardized way. Until now, the software clustering community does not have a standard way for

presenting new software clustering algorithms. Furthermore, there are no guidelines on what is the required information for a newly introduced algorithm.

- One of the biggest challenges in the algorithm improvement process is the identification of the weaknesses of an algorithm that need to be improved. To date, there is no formal process to achieve this.

This chapter addresses these problems. We introduce a method for the selection of a software clustering algorithm for specific needs. We also develop a formal description template that allows to formally specify a software clustering methodology. The adoption of the template by software clustering researchers should lead to better communication in this research field and the development of more effective software clustering approaches.

Finally, we introduce a method for the improvement of existing software algorithms. Our approach allows the evaluation of the new algorithm in earlier stages before it is fully implemented and finalized.

Four distinct case studies demonstrate the applicability and usefulness of the methods introduced in this chapter.

The structure of the rest of this chapter is as follows. Section 7.1 presents the formal description template as well as the two methods for algorithm selection and algorithm improvement. The case studies are presented in Section 7.2. Discussion on the benefits of the methods presented takes place in Section 7.3.

7.1 Describing software clustering algorithms

Part of the difficulty in comparatively evaluating software clustering algorithms stems from the fact that each algorithm makes a different set of assumptions about the task at hand, such as requiring different types of information as input. Since there is no established standard for the description of a new software clustering algorithm, it is hard to gauge the relative merits of different approaches.

In this section, we propose a consistent format for the description of software clustering algorithms, not unlike the one used for design patterns [GHJV95]. In Sections 7.1.1 and 7.1.2 respectively, we use the terminology introduced here to develop methods for the selection of appropriate algorithms, as well as the improvement of existing algorithms.

The template for the description format is as follows:

- **Algorithm Name**. While a descriptive name is preferable, algorithm naming is entirely up to the algorithm developer.

- **Algorithm Intent**. A description of the goals behind this software clustering algorithm. Examples of such goals include:

  - To decompose a software system into subsystems
  - To recover the architecture of a software system
  - To discover possible ways to restructure the software

  Explicitly stating algorithm intent can simplify selecting a software clustering algorithm for reverse engineers, as well as understanding a clustering algorithm for

software clustering researchers.

- **Factbase properties**. A description of the input that the software clustering algorithm expects (an algorithm's input is often referred to as a factbase since it contains facts extracted from the software system). For example, software clustering algorithms often expect that the factbase includes only dependencies between modules [SMDM05, TH00a, MMCG99, LZN04].

- **Clustering objective**. A description of the ideal output of the software clustering algorithm. Based on experience or heuristics, the designer of the algorithm decides what a meaningful decomposition of a software system should look like. For example, Schwanke [Sch91] concluded that a software decomposition with maximum cohesion and minimum coupling is important for software maintainers.

- **Process Description**. A brief description of the main idea behind the algorithm's implementation. For example: "ACDC creates clusters by detecting established subsystem patterns in the given software system. Software entities that are not clustered this way are assigned to the subsystem that depends the most on them".

- **Decomposition Properties**. A description of the properties of the decompositions that the software clustering algorithm creates. These properties are either a direct result of the clustering objective (e.g. Bunch creates decompositions whose value of the objective function is higher than that for other algorithms [MMCG99]) or an artifact of the algorithm's implementation (e.g. Bunch creates decompositions that are more balanced than most algorithms [WHH05]).

- **Algorithm Restrictions**. Since software clustering algorithms are heuristic algorithms, their performance may depend on the type of software system being clustered. Such dependencies need to be documented here. For example, Bunch may not be well-suited for event-driven systems [MM01b].

- **Failed assumptions**. A description of ideas that did not work very well while developing the algorithm. This information is often omitted in publications but could be invaluable to the software clustering researcher.

- **Detailed algorithm description**. The nuts and bolts of the algorithm's implementation are presented here. This will often be the only section of significant length in an algorithm description.

7.1.1 Algorithm Selection

In this section, we present a method that utilizes the notions of decomposition properties and algorithm restrictions introduced above in order to allow a reverse engineer to select an appropriate clustering algorithm for a given software system (or conversely to reject an algorithm that is not well suited for the task).

The case study presented in Section 7.2.1 presents an example of the application of the algorithm selection method presented below. The reader might find it beneficial to read this section and the case study in parallel.

The proposed algorithm selection method has three stages:

1. **Construct prototype decomposition**. A prototype decomposition is a crude, high-

level decomposition of the software system that can be constructed with a small amount of effort utilizing expert knowledge, existing documentation, and the reverse engineer's experience. It only needs to describe the top level subsystems in the software system, i.e. nested subsystems or dependencies between subsystems are not required. Several efficient methods for the construction of such a view have been presented in the literature [MJ06].

The purpose of the prototype decomposition is to stand in as a model for an authoritative decomposition. It will be used to verify the usability and applicability of decompositions created by the candidate software clustering algorithms.

2. **Apply algorithm restrictions**. The formal description of any candidate software clustering algorithm will contain a number of algorithm restrictions. If the given software system or its prototype decomposition match any of these restrictions, then the algorithm is rejected.

3. **Match decomposition properties**. A candidate software clustering algorithm is selected if the prototype decomposition can be produced by the algorithm. This means that the decomposition properties of the candidate algorithm must be consistent with the prototype decomposition, something that is usually easy to verify, either manually or with the help of existing tools. The reverse engineer needs to decide what level of discrepancy is too high for the algorithm to be selected.

Many software clustering algorithms have configuration parameters that need to be calibrated once an algorithm has been selected. The proposed method can help with this

task as well. The reverse engineer can measure the effect that changes to the configuration parameters have on the aforementioned discrepancy between the decomposition properties of the candidate algorithm and the prototype decomposition. Parameter values that minimize this discrepancy can be utilized for the clustering task.

7.1.2 Algorithm Improvement

Software clustering researchers strive to improve the effectiveness of their algorithms, either by conceiving new clustering objectives, or by improving the process that attempts to realize existing objectives. A typical researcher work flow is shown below:

1. Design a new clustering objective or a new clustering process.
2. Implement the new approach.
3. Apply the new approach to a reference system with an existing authoritative decomposition.
4. Compare the produced decomposition to the authoritative decomposition.
5. Repeat from step 1 until satisfactory results are obtained.

The main drawback of the above work flow is that an implementation is required before the new approach can be evaluated. However, the logic of a software clustering algorithm is usually complicated. As a result, the implementation phase often requires a large time investment. It would be beneficial if new clustering ideas could be evaluated without a time consuming implementation. In the following, we present a new work flow

that allows for an early evaluation of clustering ideas. As above, the reader might want to read in parallel with the case study in Section 7.2.2.

The proposed work flow has eight stages:

1. **Idea conception**.

   Based on her experience and domain knowledge, a software clustering researcher can conceive of novel clustering objectives, decomposition properties, or factbase properties that could help improve the effectiveness of existing software clustering algorithms.

2. **Reference system selection**.

   The reference software systems will be used to verify the feasibility of the new clustering idea. Their selection depends on the research goal. When the goal is to evaluate a new clustering objective, various types of software systems should be selected as reference systems. When the goal is to evaluate new approaches to achieve existing clustering objectives, then the authoritative decomposition of the selected reference systems has to comply with the stated clustering objective.

3. **Authoritative decomposition construction**.

   If an authoritative decomposition exists for a reference system, then it can be used directly. Otherwise, a prototype decomposition needs to be constructed as explained in Section 7.1.1.

4. **Idea verification**.

The purpose of this stage is to determine whether the authoritative decompositions of the selected reference systems have properties that reflect the clustering idea being considered. This can be done either manually or by developing a verification application. In either case, the amount of time required for this stage should be significantly less than fully implementing and testing the clustering idea. As a result, a researcher can evaluate the feasibility of the new clustering idea much earlier.

5. **Verification result analysis**.

   If the verification stage determined that the reference systems are compatible with the clustering idea, then the full implementation of the idea can commence in the next stage. However, if an incompatibility was detected, then one of the following two actions should be taken:

   (a) A new set of reference systems is selected and the verification process repeated. If successful, then implementation can begin, but the approach's final documentation will need to include an algorithm restriction documenting the fact that this approach is not appropriate for all software systems.

   (b) The new clustering idea is rejected. In this case, the "failed assumptions" section of the approach's documentation needs to include this negative result.

6. **Algorithm implementation**.

   The full implementation of the new clustering idea takes place during this stage.

7. **Algorithm evaluation**.

The effectiveness of the new clustering idea can be evaluated by comparing its results to the results of existing approaches by using established evaluation criteria, such as MoJo [WT03] or the Koschke-Eisenbarth measure [KE00].

8. **Final documentation**.

   The last stage of our proposed work flow requires that a formal description of the new clustering approach is created following the description template presented in Section 7.1. This way the software clustering research community can be made aware of all results, both positive and negative, obtained during this process.

The following section presents four case studies that follow this work flow and demonstrate its usefulness.

7.2 Case Studies

In this section, we present examples of applying the methods for algorithm selection (Section 7.2.1) and algorithm improvement (Sections 7.2.2 to 7.2.4) presented earlier in Sections 7.1.1 and 7.1.2 respectively. Each case study follows the exact steps outlined by the corresponding method.

7.2.1 Algorithm Rejection

In this case study, our goal is to decompose version 6.3 of Vim, a well-known text editor, into subsystems. In our attempt to select an appropriate clustering algorithm for this

process, we applied the algorithm selection method presented in Section 7.1.1 with ACDC and Bunch as the candidate algorithms.

Construct prototype decomposition. We constructed a prototype decomposition for Vim by consulting documentation and the directory structure. The prototype decomposition contained 51 entities.

Apply algorithm restrictions. While none of Bunch's algorithm restrictions applied, ACDC is not meant for systems with a small number of entities. Since 51 can be considered small for clustering purposes, ACDC could be rejected at this point. However, for the purposes of demonstration, we will consider ACDC in the next step as well.

Match decomposition properties. The relevant decomposition property of Bunch is a high value of the objective function MQ [MMCG99]. We calculated the value of the MQ function for the prototype decomposition at 0.33, a significantly low value (see the case study in Section 7.2.4 for more typical values of MQ). This means that the Bunch algorithm will fail to construct an acceptable decomposition for Vim, and is therefore rejected.

The relevant decomposition property of ACDC is that a large number of cluster contains a "dominator node" [TH00a]. We found out that in the prototype decomposition there are only 4 out of 10 clusters that have a "dominator node". This leads to the rejection of the ACDC algorithm as well, though the decision here was closer.

We checked our previous conclusions by actually using Bunch and ACDC to create decompositions of Vim. Both decompositions were significantly different from our pro-

totype. The MoJoFM value for Bunch was 13.64% and for ACDC 28.26%. Typical values for successful algorithms are between 50% and 75%.

7.2.2 Body - Header Processing

The goal of this case study is to develop a concept that will improve the decompositions that any software clustering algorithm creates when applied to systems developed in C or C++.

Idea conception. A typical implementation of a software system written in C has two types of files:

1. Header file (.h) - a file that contains declarations

2. Body File (.c) - a file that contains implementations

Several research studies have identified the strong relation between body and header files. For instance, the ACDC algorithm defines a body-header pattern, that assigns a header file and a body file into a cluster [TH00a]. Examination of manually created authoritative decompositions of well-known systems confirms that body and header files belong in the same cluster. We want to investigate whether automatically assigning body and header files together will improve the effectiveness of existing algorithms.

Reference system selection. We selected the Linux and Mozilla software systems (both written in C or C++).

Authoritative decomposition construction. Fortunately, authoritative decompositions for both systems have been created by the research community. The authoritative decomposition for this version of the Linux kernel was presented in [BHB99]. An authoritative decomposition for this version of Mozilla was presented in [Xia04].

Idea verification. To verify that our idea has merit we needed to check that it holds true in the authoritative decompositions of our reference systems. A small program was developed to find and print the names of header files that belong to a different cluster than the corresponding body files. No such header files were found (there were however 6 header files in Linux and 10 in Mozilla that could not be matched to a body file).

Verification result analysis. The verification results show that our assumption that body and header files should go together is a good one. Before implementing this concept, we checked whether it would have any practical advantage for ACDC and Bunch. Table 7.1 shows the number of misplaced header files in decompositions produced by ACDC and Bunch for both reference systems. There is clearly a lot of room for improvement.

System	ACDC	Bunch
Mozilla	360	44
Linux	94	45

Table 7.1: Number of misplaced header files in automatically created decompositions

This experiment revealed a bug in the ACDC implementation as well, since ACDC's

body-header pattern should have ensured that the numbers in the above table are significantly smaller. The bug was identified and fixed.

Algorithm implementation. To minimize the modification of existing clustering algorithms, we introduced a pre-processing and a post-processing step to every algorithm.

The pre-processing step removes all header files that have corresponding body files from the factbase and keeps track of the associated body and header files. As a result, after the pre-processing step the modified factbase does not contain entities corresponding to the header files.

The post-processing step assigns all header files to the same cluster as their corresponding body files.

Algorithm evaluation. In order to evaluate whether the modified algorithms are more effective, we calculated the MoJoFM values for both algorithms with and without the pre- and post-processing steps. Table 7.2 shows the obtained results. It is clear that the effectiveness of the two algorithms has improved.

System + Algorithm	Before	After
ACDC + Linux	61.92	69.70
ACDC + Mozilla	59.68	62.76
Bunch + Linux	69.62	74.58
Bunch + Mozilla	60.30	62.43

Table 7.2: MoJoFM values for ACDC and Bunch before and after modification

Final documentation. It is easy to modify the formal description of both algorithms to include the two extra processing steps.

It is interesting to note that this approach also makes these algorithms more scalable, since it decreases the number of entities they have to deal with. Furthermore, this concept can be generalized to other programming languages as long as the language distinguishes between an interface definition and its implementation.

7.2.3 Component - Library Processing

This case study attempts to utilize containment information related to components and libraries in order to obtain better software decompositions. Since components and libraries are treated similarly in this case study, the term "component" will refer to either a component or a library.

Idea conception. A typical modern software system contains a number of components. A reverse engineer can gather information about the system's components by applying utilities that analyze makefiles such as Makefile::parser [mak], as well as utilities that analyze binary modules such as Dependency Walker [dep]. We will attempt to leverage such information about components in order to improve the effectiveness of any software clustering algorithm.

Reference system selection. We selected the same version of Mozilla as in the previous case study. It is a system that contains a large number of components.

Authoritative decomposition construction. The same authoritative decomposition as in the previous case study was used.

Idea verification. Once again, we consulted the authoritative decomposition to determine whether our idea holds true, i.e. the entities of a given component are clustered together. We performed this verification phase for all components of Mozilla. For each component, we identified a list of relevant modules and determined whether they belong in the same cluster.

Verification result analysis. The verification stage confirmed that all modules of a component belong to the same cluster. We did find that some clusters of the authoritative decomposition have more members than the corresponding components. This can be attributed either to entities that were not compiled when we extracted the list of modules for the component, or to the inclusion of unit test modules that are not part of the component. However, our main idea was clearly verified.

Algorithm implementation. The previous analysis suggests that a software clustering algorithm should automatically assign all modules from a component in one cluster and then decompose each cluster separately. This should improve both the scalability of the algorithm as well as the quality of the produced decomposition. We suggest the following algorithm for the construction of a software decomposition:

1. Construct a factbase whose entities are components, as well as modules which are not part of components.

2. Execute the software clustering algorithm producing decomposition A

3. When the size of a component is small, then assign all modules from the component to a cluster that will replace the entity representing the component. Otherwise, perform the following operations

   (a) Construct a factbase whose entities are the modules of the component. This factbase may still contain components, i.e. sub-components of the original component.

   (b) Execute the software clustering algorithm on the component factbase creating decomposition B. Then, merge B into A. This will eventually create a nested decomposition. If that is not desirable, the clusters of B can directly replace the component they correspond to in A creating a flat decomposition.

This process requires iterative execution of a software clustering algorithm. However, every execution constructs a decomposition for a significantly smaller number of entities than the single execution of the original process. Therefore, the new process is more scalable than the original one.

Algorithm evaluation. Results of the newly modified algorithm can be evaluated in a manner similar to the previous case study. In order to evaluate whether the modified algorithms are more effective, we calculated the MoJoFM values for both algorithms with and without modification. Table 7.3 shows the obtained results. It is clear that the effectiveness of the two algorithms has improved.

System + Algorithm	Before	After
ACDC + Mozilla	59.68	75.71
Bunch + Mozilla	60.30	66.22

Table 7.3: MoJoFM values for ACDC and Bunch before and after modification

Final documentation. The formal description of any algorithm can be modified to include the iterative process.

7.2.4 Improving Bunch

In this last case study, we investigate ways to improve Bunch.

Idea conception. The Bunch algorithm can be improved by either developing a new objective function or improving the search heuristics. The main decomposition property of Bunch is a high value for its objective function (MQ). If authoritative decompositions do not have similarly high values, then this might indicate that research efforts may need to be directed towards finding a different objective function.

Reference system selection. We selected Mozilla as a reference system because it is the largest and most complex system for which an authoritative decomposition was readily available.

Authoritative decomposition construction. As mentioned in the previous case studies, an authoritative decomposition exists for Mozilla. Since Bunch produces flat decomposi-

tions (no nested subsystems) we transformed the authoritative decomposition of Mozilla to a flat version as well (see Section 4.1).

Idea verification. We calculated the MQ function for the authoritative decomposition as well as the one produced by Bunch. The two decompositions contained a similar number of clusters (10 for the authoritative decomposition, 15 for Bunch's output). However, the authoritative decomposition MQ value was 4, while that produced by Bunch had a value of 8.

Verification result analysis. The value of the MQ function for the authoritative decomposition of Mozilla is significantly lower than that for the decomposition produced by Bunch. This would seem to indicate that efforts should be directed towards developing a different objective function.

To be more conclusive, we repeated the previous steps on a subsystem of Mozilla. The value of the MQ function for the authoritative sub-decomposition was 4.1, while the corresponding decomposition created by Bunch had a value of 6.8.

These results support the conclusion that a new objective function would improve the effectiveness of Bunch. Harman et al. [HSM05] came to a similar conclusion in their empirical study of Bunch's effectiveness.

Developing such an objective function however, is beyond the scope of this thesis. As a result, the remaining steps of the method are omitted.

7.3 Discussion

In this section, we outline some of the advantages of using the formal description template as well as the algorithm selection and improvement methods presented in this chapter.

Selecting strategies for algorithm improvement. Suppose that a researcher is interested in improving her software clustering algorithm but is unsure how to proceed, i.e. which aspect of the algorithm to optimize. The researcher should select different types of software systems with authoritative decompositions that were created based on the current clustering objective. If the decomposition properties of the algorithm are not found in the authoritative decompositions, then effort must be spent to improve aspects of the algorithm that are responsible for the decomposition properties. The last case study was an example of this approach.

Mixing algorithms. The decomposition of a software clustering algorithm into clustering objectives, factbase properties and decomposition properties gives researchers the freedom to mix such elements from different algorithms. When a software clustering algorithm is considered for improvement it can possibly address some of its weaknesses by borrowing strong elements from other algorithms (it can also avoid pitfalls by studying failed assumptions of other algorithms).

Evaluating software clustering without authoritative decompositions. Almost all clustering evaluation methods in the literature require an authoritative decomposition. In its

absence, evaluating clustering results is practically impossible, partly because it is hard to identify what the correct decomposition for a given clustering objective is. When every software clustering algorithm will have a more formal definition of the decomposition properties it aims to produce, it will be easier to develop methods to compare algorithms. For instance, algorithms that construct software decomposition based on the principle of high cohesion and low coupling can be compared by measuring how well they comply to that principle.

Verification of implementation. After a software clustering algorithm is implemented, it has to be tested. The testing of a software clustering algorithm is difficult because it is often hard to know whether an unsuccessful result is due to an implementation bug or to an inappropriate clustering objective. When decomposition properties have been identified, then a researcher has new tools to verify the implementation of a software clustering algorithm. When the produced decomposition does not comply with the decomposition properties, that indicates a bug in the implementation. The ACDC bug found during the body-header case study is an example of this.

Development of new types of software decompositions. The formal description template introduced in this chapter allows for the separation between developing a new type of software decomposition and developing a software clustering algorithm that produces such a decomposition. This separation may encourage other members of the reverse engineering community to develop new types of software decomposition without being experts on software clustering. This has the potential to introduce fresh ideas into this

research field.

Shared failed assumptions. Most of the publications on software clustering algorithm describe success stories. However, there are often many failures that are left undocumented since they did not lead to successful results. This knowledge could be very useful to other researchers. We hope that the "failed assumptions" section in the formal description template will encourage the sharing of such information.

This chapter presented a formal description template for software clustering algorithms. Using this template, we introduced two methods for algorithm selection and algorithm improvement. Four case studies demonstrated the benefits of utilizing these methods in real-life scenarios. Finally, we also developed two generic approaches for improving the scalability and efficiency of software clustering algorithms.

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

8 Conclusions

In today's high-technology world, every one of us depends on the correct operation of software products for a number of daily activities. Marketing demands dictate that new complicated features are delivered within a short amount of time. Highly customizable solutions that are able to interoperate with existing software are often required by customers. As a result, software development teams are often asked to deliver high quality, flexible, customizable, and standards-compliant software within a limited timeframe.

This list of factors often results in software that is very complex. This complexity means that the cost of development and maintenance is also very high, since large amounts of time and effort are spent in trying to understand the software system. The program comprehension research community has developed a number of different approaches that aim to help developers comprehend software systems faster and more accurately. Several of these approaches involve software clustering.

Software clustering is a research area that has been developing for a long time. Many different software clustering algorithms have been developed. These algorithms have been proposed in the literature in an effort to address various reverse engineering issues including reflexion analysis, software evaluation and information recovery.

The successful development of efficient software clustering algorithms depends on achievements in evaluation theory. Therefore, our main contribution to the research community is improving evaluation theory. We presented two methods for the evaluation of nested decompositions. These are the only methods in the software clustering literature that address this problem. We addressed another open question related to software clustering, that of the comparability of software clustering algorithms. Also, we developed a novel set of structure indicators that can augment the evaluation picture provided by traditional comparison methods, such as MoJoFM and KE. Finally, we presented a new process for the evaluation of software clustering algorithms by utilizing multiple simulated authoritative decompositions.

We also addressed issues regarding the practical application of software clustering algorithms. We developed a method for the selection of a software clustering algorithm for specific needs. We proposed a formal description template that allows to formally specify a software clustering methodology. The adoption of the template by software clustering researchers should lead to a more efficient deployment of software clustering algorithms.

The following section explains our research contributions in more detail.

8.1 Research contributions

1. Nested Decomposition Evaluation. We introduced two methods that measure similarity between two nested decompositions of the same software system. These

methods can determine whether a nested decomposition produced by a given algorithm comes close to the authoritative decomposition constructed by a system expert. Also, these methods are useful in many scenarios including selection and calibration of software clustering algorithms.

2. Structure Evaluation. We developed a novel set of indicators that evaluate structural discrepancies between two decompositions of the same software system. The developed set of structure indicators helps a software maintainer better understand the properties of automatically created decompositions. They also allow researchers to investigate the differences between existing evaluation approaches in a reduced search space.

3. Comparability. We introduced and studied the notion of comparability of software clustering algorithms and we presented a measure that gauges it. Comparability between a newly introduced and known algorithm is important for an unbiased evaluation of the new algorithm. This dissertation presented the first extensive study on the comparability of software clustering algorithms.

4. MoJoFM vs KE. We investigated possible reasons for the discrepancies between two of the more popular evaluation methods, MoJoFM and KE. The presented study gives theoretical justification and experimental evidence that the two evaluation methods have fundamental differences despite their common goal.

5. LimSim. We proposed a new process for the evaluation of clustering algorithms by utilizing multiple simulated decompositions. The new process allows for more

accurate evaluation of the efficiency of a software clustering algorithm, since it employs multiple simulated authoritative decompositions.

6. Algorithm Selection and Improvement. We presented a method for the selection of a software clustering algorithm for specific needs. This method provides guidelines for the reverse engineer on how to select a suitable algorithm for their task. We also introduced a method for improving software clustering algorithms. The new method helps reduce algorithm improvement time, because new ideas can be verified without full implementation. Four case studies demonstrated the benefits of utilizing these methods in real-life scenarios. In two of the case studies, the improvements we investigated resulted in increased scalability for the clustering algorithm.

7. Formal Description. We introduced a formal description template that allows to formally specify a software clustering methodology. The adoption of the template by software clustering researchers should lead to better communication in this research field and the development of more effective software clustering approaches. In addition, the proposed format was used for presenting several known software clustering algorithms. Their descriptions are available online [wik] as part of a community repository for software clustering methodology.

8. Generators. We introduced tools that generate artifacts that can be used to study the behaviour of existing approaches for the comprehension of large software systems. We also presented three distinct applications of the introduced tools: the development of a simple evaluation method for clustering algorithms, the study of

the behaviour of the objective function of Bunch, and the calculation of a congruity measure. The presented tools were extensively used during the course of our research.

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

8.2 Future Research Directions

Additional experiments. Each of the contributions of this thesis 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, software clustering algorithms and evaluation measures, and to determine whether we encounter similar behaviour or not.

Performance. We performed several experiments to validate the proposed approaches in this thesis. However, we did not test the performance of these approaches. Work needs to be done in order to determine whether the performance of the current implementations is suitable for their task. Also, it would be beneficial to optimize the performance of the created tools.

Generalization of UpMoJo. We developed UpMoJo distance in order to evaluate nested decompositions in a lossless fashion. UpMoJo expands on MoJo distance by adding an operation that is suitable for nested decompositions. It would be interesting to see if other

measures, such as KE, can be adapted similarly.

Using labels to evaluate nested decompositions. The evaluation methods for nested decompositions that we developed do not consider labelling. However, labelling is an important factor for reverse engineers, since good labels can help significantly in the understanding of large software systems, while bad ones reduce the usability of software decompositions. Therefore, further work needs to be done in order to incorporate the quality of cluster labels into the assessment of the quality of nested software decompositions.

Evaluate factbase quality. The typical goal of evaluation methods is to calculate the quality of software clustering methods. Until now, there is no evaluation method whose goal is to assess the quality of a factbase. We know that the quality of a factbase can have a significant effect on the outcome of software clustering algorithms. Therefore, the evaluation of factbase quality would help researchers develop more sophisticated fact extraction algorithms.

Expansion of LimSim. We developed a process that allows the evaluation of software clustering algorithms on multiple simulated authoritative decompositions. The process is based on operations that are designed in such a way as to ensure that the resulting decomposition is indeed authoritative for the resulting MDG.

The current process produces simulated authoritative decompositions without consideration for the different types of software systems. It would be interesting to develop

new operations for LimSim in such a way that the constructed simulated authoritative decompositions will belong to a specific type of software system. Such a process would help researchers assess the quality of software clustering algorithms on different types of software systems.

Extended Indicators. We introduced a set of indicators that measure structure discrepancies between two decompositions. It would be desirable to develop other indicators for the measurement of other decomposition properties, such as a set of indicators that will measure labelling differences between two decompositions of the same software system.

Reduced space comparability. The main corollary from our introduction of the notion of comparability is that evaluating algorithms against an authoritative decomposition should only be done between comparable algorithms.

It would be interesting to develop a methodology for the evaluation of non-comparable software clustering algorithms in a reduced space. In this space, all evaluated algorithms would behave as if they are comparable.

END Extension. One of the advantages of END is its ability to evaluate software clustering algorithms using different weighting schemes. The weighting schemes allow us to analyze the quality of a decomposition at different granularity levels. Currently, the END framework provides a small number of weighting schemes. Work needs to be done to develop new weighting schemes for the END framework, as well as develop formal guidelines for the usage of the END framework.

MoJoFM vs KE. We have found that two of the more popular evaluation methods, MoJoFM and KE, are significantly different. However, we know that there are many research studies using these two methods. We would like to develop an approximate translation table from MoJoFM to KE and vice versa. Obviously, such a translation will not be 100% accurate, but it will give intuition in terms of comparing evaluation results.

Matching evaluation method to objectives. It would be interesting to develop a method that helps answer the question of which evaluation method is better in the context of specific clustering objectives. This tool would help reverse engineers identify the most suitable evaluation measure for their needs.

Formal description. We used the formal description of a clustering algorithm to present several known software clustering algorithms. It would be important to extend this to include as many algorithms as possible.

Factbase improvements. We presented case studies in which the quality of software clustering algorithms was improved by improving the quality of the factbase used. This is evidence of the importance of combining information from various sources. Further research is required to determine the most effective way to do this.

Algorithm Selection. We proposed a method for the selection of a software clustering algorithm. It would be interesting to develop an interactive system which can select a suitable software clustering algorithm. The system will be an expert type system that

contains a large database of existing software clustering algorithms and automatically recommends the most suitable algorithm for a specific task. In addition, the system could provide calibration parameters for the selected algorithm.

8.3 Closing remarks

Our work contributes to the field of software engineering by improving the evaluation of software clustering algorithms, introducing methods for algorithm selection and improvement, as well as addressing several related research challenges. We presented experimental data that demonstrate that our techniques were effective in solving various research issues.

We also created a wiki [wik], that contains information about software clustering methodology. The wiki enables authorized users to maintain information about new discoveries in software clustering. We hope that the wiki will become a community repository, which contains up-to-date information about software clustering algorithms, evaluation methods, factbases and tools. This should improve collaboration between software engineers and researchers in the software clustering field.

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

A Examples of algorithm templates

Chapter 7 introduced a formal description template for software clustering algorithms. This appendix presents formal descriptions of several well known software clustering algorithms, such as ACDC, Bunch, LIMBO, and Agglomerative algorithms.

A.1 ACDC Algorithm

The ACDC algorithm has been developed by Vassilios Tzerpos [TH00a] and has demonstrated promising results on several large software systems. In this section, we will present the formal description for the ACDC algorithm.

Algorithm Intent

ACDC produces flat or nested decompositions of a software system. The algorithm assigns meaningful names to the clusters it creates.

Factbase Properties

The factbase includes only dependencies between entities.

Clustering Objectives

The produced decomposition has the following properties:

1. Effective cluster names

2. Bounded cluster cardinality

3. File-level granularity

Process Description

ACDC creates clusters by detecting established subsystem patterns in the given software system. Software entities that are not clustered this way are assigned to the subsystem that depends the most on them.

Decomposition Properties

1. The cluster name is one of the following options:

   - Directory name

   - Module name with large out-degree

   - "Support" for the cluster that contains all utilities

   - Module name of dominator node

2. All utilities modules belong to one cluster

3. A large number of clusters contain a dominator node

Algorithm Restrictions

The algorithm is not suitable for systems with less than 100 source files [TH00a].

Failed Assumptions

Detailed Algorithm Description

ACDC produces a software decomposition from a software system in stages:

1. Skeleton construction
2. Orphan Adoption

The skeleton construction phase creates a skeleton of the final decomposition by identifying subsystems using a pattern-driven approach. Depending on the pattern used, the subsystems are given names. In the second stage, the algorithm completes the decomposition by using the Orphan Adoption algorithm.

Skeleton construction

The skeleton construction phase has seven steps:

1. Apply the source file pattern
2. Apply the body-header pattern
3. Create the potential dominator list
4. Create a support library and dispatcher list

5. Create initial skeleton

6. Iterate the sub-graph dominator pattern including modules from the support library and dispatcher list

7. Apply both the support library pattern and the dispatch pattern to modules from corresponding list

All steps that apply patterns are straight forward, grouping entities according to the pattern rule. The other steps are described below.

Create the potential dominator list

This creates a list of all modules in the systems. The list is sorted in order of ascending out-degree, i.e. the first element in the list is the one with the smallest number of outgoing edges.

Create a support library and dispatcher list

ACDC removes from the potential dominator list, all modules that have an in- or out-degree larger than 20. These nodes are placed in two lists:

- Support library list. This list contains all modules with an in-degree larger than 20.
- Dispatcher list. This list contains all modules with an out-degree larger than 20.

Create initial skeleton

ACDC goes through each module n in the potential dominator list and examines whether n is the dominator node of a subsystem, according to the sub-graph dominator pattern.

When a dominator node is found, 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.

After all modules from the potential dominator list have been examined, ACDC organizes the obtained subsystems, so that the containment hierarchy is a tree.

Orphan Adoption

In the previous phase, the skeleton of a software decomposition was constructed. Some modules are left unassigned from the first phase. In this phase, the algorithm assigns all non-clustered modules to an already created subsystem, in particular the subsystem that depends the most on the unassigned module.

A.2 Bunch

Bunch is a clustering suite that has been developed by Spiros Mancoridis [MMCG99] and his research team at Drexel University. It has been presented in a number of publications [MMR+98, MM06]. We developed the following formal description for Bunch.

Algorithm Intent

Bunch produces flat decompositions of a software system that exhibit high cohesion and low coupling.

Factbase Properties

The factbase is a module dependency graph (MDG).

Clustering Objectives

The Bunch algorithm constructs software decompositions with maximum cohesion and minimum coupling.

Process Description

The Bunch algorithm is an optimization search algorithm that finds a partition with the maximum value of the MQ objective function.

Decomposition Properties

The Bunch decomposition has a high value for its objective function MQ.

Algorithm Restrictions

The Bunch may not be well-suited for event-driven systems [MM01b].

Failed Assumptions

Detailed Algorithm Description

Bunch starts by generating a random partition of the module dependency graph. Then, entities from the partition are regrouped systematically by examining neighbouring partitions in order to find a better partition. When an improved partition is found, the process repeats, i.e. the found partition is used as the basis for finding the next improved partition. The algorithm stops when it cannot find a better partition.

A.3 Limbo

The algorithm was developed by Periklis Andritsos and Vassilios Tzerpos [AT05].

Algorithm Intent

To generate decompositions that exhibit the least information lost when entities are represented by their clusters.

Factbase Properties

The factbase is a generic dependency data table where each row describes one entity to be clustered. Each column contains the value for a specific attribute.

Clustering Objectives

The Limbo algorithm produces software decompositions with minimum information loss.

Process Description

An agglomerative clustering algorithm that on each step merges the two clusters with the least information loss.

The information lost is calculated using the Agglomerative Information Bottleneck algorithm.

Decomposition Properties

The LIMBO decomposition has a small value for its information lost function.

Algorithm Restrictions

Failed Assumptions

Detailed Algorithm Description

LIMBO has four phases:

1. Creation of the Summary Artefacts
2. Application of the AIB algorithm
3. Associating original artefacts with clusters

4. Determining the number of clusters

A.4 Agglomerative Clustering Algorithms

Agglomerative clustering algorithms [MB04] are well known clustering algorithms that have been extensively studied. Even though they are designed for any data set, they are widely used to cluster software systems. Following is the formal description of agglomerative clustering algorithms.

Algorithm Intent

The goal of the agglomerative algorithms is to discover hierarchical relations between entities.

Factbase Properties

The factbase is a generic dependency data table where each row describes one entity to be clustered. Each column contains the value for a specific attribute.

Clustering Objectives

The clustering objective is dependent on the update rule. For instance, complete linkage gives the most cohesive clusters. Clusters formed using single linkage approach are not cohesive as complete linkage, and the results of average linkage algorithms lie somewhere between those of single and complete linkage.

Process Description

Agglomerative algorithms start at the bottom of the hierarchy by iteratively grouping similar entities into clusters. At each step, the two clusters that are most similar to each other are merged and the number of clusters is reduced by one.

Decomposition Properties

Algorithm Restrictions

Failed Assumptions

Detailed Algorithm Description

Agglomerative algorithms perform the following steps:

1. Compute a similarity matrix
2. Find the two most similar clusters and join them
3. Calculate the similarity between the joined clusters and others obtaining a reduced matrix
4. Repeat from step 2 until two clusters are left

Bibliography

[AATW07] Bill Andreopoulos, Aijun An, Vassilios Tzerpos, and Xiaogang Wang. Clustering large software systems at multiple layers. *Information & Software Technology*, 49(3):244–254, 2007.

[acd] `http://www.cse.yorku.ca/~bil/downloads/`.

[ACL00] William Aiello, Fan Chung, and Linyuan Lu. A random graph model for power law graphs. *Experimental Math*, 10:53–66, 2000.

[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] N. Anquetil and T.C. Lethbridge. Experiments with clustering as a software remodularization method. In *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, pages 235–255, 6-8 Oct. 1999.

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

[AT03] Periklis Andritsos and Vassilios Tzerpos. Software clustering based on information loss minimization. In *Proceedings of the Tenth Working Conference on Reverse Engineering*, pages 334–344, November 2003.

[AT05] Periklis Andritsos and Vassilios Tzerpos. Information-theoretic software clustering. *IEEE Trans. Softw. Eng.*, 31(2):150–165, 2005.

[BHB99] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux as a case study: Its extracted software architecture. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.

[BS91] Rodrigo A. Botafogo and Ben Shneiderman. Identifying aggregates in hypertext structures. In *HYPERTEXT '91: Proceedings of the third annual ACM conference on Hypertext*, pages 63–74, New York, NY, USA, 1991. ACM Press.

[BT04] M. Bauer and M. Trifu. Architecture-aware adaptive clustering of oo systems. In *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*, pages 3–14, 2004.

[bun] `http://serg.cs.drexel.edu/redmine/projects/list_files/bunch`.

[Bus97] Samuel Buss. Alogtime algorithms for tree isomorphism, comparison, and canonization. In *In Computational Logic and Proof Theory, 5th Kurt Godel Colloquium'97, Lecture Notes in Computer Science*, pages 18–33. Springer-Verlag, 1997.

[CC05] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *Software Metrics, 2005. 11th IEEE International Symposium*, page 9pp., 19-22 Sept. 2005.

[CCK00] Gerardo Canfora, Jörg Czeranski, and Rainer Koschke. Revisiting the delta ic approach to component recovery. In *WCRE '00: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, page 140, Washington, DC, USA, 2000. IEEE Computer Society.

[CDH+03] J. Clarke, J.J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *Software, IEE Proceedings -*, 150(3):161–175, 2003.

[CKS07] Andreas Christl, Rainer Koschke, and Margaret-Anne Storey. Automated clustering to support the reflexion method. *Inf. Softw. Technol.*, 49(3):255–274, 2007.

[CMPS07] Giulio Concas, Michele Marchesi, Sandro Pinna, and Nicola Serra. Power-laws in a large object-oriented software system. *IEEE Trans. Softw. Eng.*, 33(10):687–708, 2007.

[Con68] M.E. Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968.

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

[CSOT08] Sylvain Chardigny, Abdelhak Seriai, Mourad Oussalah, and Dalila Tamzalit. Extraction of component-based architecture from object-oriented systems. *Software Architecture, Working IEEE/IFIP Conference on*, 0:285–288, 2008.

[DB00] J. Davey and E. Burd. Evaluating the suitability of data clustering for software remodularisation. In *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, pages 268–276, 23-25 Nov. 2000.

[dep] `http://www.dependencywalker.com/`.

[Dha95] Harpal Dhama. Quantitative models of cohesion and coupling in software. In *Selected papers of the sixth annual Oregon workshop on Software metrics*, pages 65–74, New York, NY, USA, 1995. Elsevier Science Inc.

[Dmi04] Mikhail Dmitriev. Profiling java applications using code hotswapping and dynamic call graph revelation. *SIGSOFT Softw. Eng. Notes*, 29(1):139–150, 2004.

[DMM99] D. Doval, S. Mancoridis, and B.S. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *Software Technology and Engineering Practice, 1999. STEP '99. Proceedings*, pages 73–81, 30 Aug.-2 Sept. 1999.

[DTD01] Demeyer, Tichekaar, and Ducasse. Famix 2.1 - the famoos information exchange model. technical report. Technical report, Univ. of Bern, 2001.

[DYM$^+$08] Jens Dietrich, Vyacheslav Yakovlev, Catherine McCartin, Graham Jenson, and Manfred Duchrow. Cluster analysis of java dependency graphs. In *SoftVis '08: Proceedings of the 4th ACM symposium on Software visualization*, pages 91–94, New York, NY, USA, 2008. ACM.

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

[Fal98] Emanuel Falkenauer. *Genetic Algorithms and Grouping Problems*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[FPG03] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 23–32, 22-26 Sept. 2003.

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

[GK00] Jean-Fran&ccdedil;ois Girard and Rainer Koschke. A comparison of abstract data types and objects recovery techniques. *Sci. Comput. Program.*, 36(2-3):149–181, 2000.

[GL70] I. Gitman and M.D. Levine. An algorithm for detecting unimodal fuzzy sets and its application as a clustering technique. *Transactions on Computers*, C-19(0018-9340):583–593, 1970.

[GL00] Michael W. Godfrey and Eric H. S. Lee. Secrets from the monster: Extracting mozilla's software architecture. In *Second International Symposium on Constructing Software Engineering Tools*, June 2000.

[GMY04] Michel L. Goldstein, Steven A. Morris, and Gary G. Yen. Problems with fitting to the power-law distribution, August 2004.

[Gow71] J. C. Gower. A general coefficient of similarity and some of its properties. *Biometrics Journal*, 27:857–874, 1971.

[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.

[HH00] A.E. Hassan and R.C. Holt. A reference architecture for web servers. In *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, pages 150–159, 23-25 Nov. 2000.

[HH04] A.E. Hassan and R.C. Holt. Studying the evolution of software systems using evolutionary code extractors. In *Software Evolution, 2004. Proceedings. 7th International Workshop on Principles of*, pages 76–81, 2004.

[HLBAL05] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering behavioral design models from execution traces. In *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*, pages 112–121, 21-23 March 2005.

[HMY06] Gang Huang, Hong Mei, and Fu-Qing Yang. Runtime recovery and manipulation of software architecture of component-based systems. *Automated Software Engg.*, 13(2):257–281, 2006.

[Hol98] R.C. Holt. Structural manipulations of software architecture using tarski relational algebra. In *Reverse Engineering, 1998. Proceedings. Fifth Working Conference on*, pages 210–219, 12-14 Oct. 1998.

[HSM05] Mark Harman, Stephen Swift, and Kiarash Mahdavi. An empirical study of the robustness of two module clustering fitness functions. In Hans-Georg Beyer and Una-May O'Reilly, editors, *GECCO*, pages 1029–1036. ACM, 2005.

[HSSW06] Richard C. Holt, Andy Schürr, Susan Elliott Sim, and Andreas Winter. Gxl: a graph-based standard exchange format for reengineering. *Sci. Comput. Program.*, 60(2):149–170, 2006.

[JD88] Anil K. Jain and Richard C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[jre] `https://wiki.cse.yorku.ca/project/cluster/tools`.

[KCK99] J. Korn, Yih-Farn Chen, and E. Koutsofios. Chava: reverse engineering and tracking of java applets. In *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, pages 314–325, 6-8 Oct. 1999.

[KDG05] A. Kuhn, S. Ducasse, and T. Girba. Enriching reverse engineering with semantic clustering. In *Reverse Engineering, 12th Working Conference on*, page 10pp., 7-11 Nov. 2005.

[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.

[KGMI04] S. Kawaguchi, P.K. Garg, M. Matsushita, and K. Inoue. Mudablue: an automatic categorization system for open source repositories. In *Software Engineering Conference, 2004. 11th Asia-Pacific*, pages 184–193, 30 Nov.-3 Dec. 2004.

[Kos00] R Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Stuttgart University, 2000.

[KR90] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley, 1990.

[LD03] Michele Lanza and Stéphane Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Trans. Softw. Eng.*, 29(9):782–795, 2003.

[LG95] A. Lakhotia and J.M. Gravley. Toward experimental evaluation of subsystem classification recovery techniques. In *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*, pages 262–269, 14-16 July 1995.

[LLG06] M. Lungu, M. Lanza, and T. Girba. Package patterns for visual architecture recovery. In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, volume 00, page 10pp., 22-24 March 2006.

[LN97] Danny B. Lange and Yuichi Nakamura. Object-oriented program tracing and visualization. *Computer*, 30(5):63–70, 1997.

[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.

[Lut01] Rudi Lutz. Evolving good hierarchical decompositions of complex systems. *J. Syst. Archit.*, 47(7):613–634, 2001.

[LZN04] Chung-Horng Lung, Marzia Zaman, and Amit Nandi. Applications of clustering techniques to software partitioning, recovery and restructuring. *J. Syst. Softw.*, 73(2):227–244, 2004.

[mak] `http://perl.enstimac.fr/perl5.6.1/site_perl/5.6.1/Makefile/Parser.html`.

[MB04] O. Maqbool and H.A. Babri. The weighted combined algorithm: a linkage algorithm for software clustering. In *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*, pages 15–24, 2004.

[MGB74] Alexander M. Mood, Franklin A. Graybill, and Duane C. Boes. *Introduction to the Theory of Statistics*. McGraw-Hill Companies, 3 edition, April 1974.

[MHH03] Kiarash Mahdavi, Mark Harman, and Robert Mark Hierons. A multiple hill climbing approach to software module clustering. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 315, Washington, DC, USA, 2003. IEEE Computer Society.

[Mit02] Brian Scott Mitchell. *A heuristic search approach to solving the software clustering problem*. PhD thesis, Drexel University, 2002. Adviser-Spiros Mancoridis.

[MJ06] Nenad Medvidovic and Vladimir Jakobac. Using software evolution to focus architectural recovery. *Automated Software Engg.*, 13(2):225–256, 2006.

[MJS+00] Hausi A. Muller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne D. Storey, Scott R. Tilley, and Kenny Wong. Reverse engineering: a roadmap. In *ICSE — Future of SE Track*, pages 47–60, 2000.

[MM01a] Brian S. Mitchell and Spiros Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *Proceedings of the International Conference on Software Maintenance*, pages 744–753, November 2001.

[MM01b] Brian S. Mitchell and Spiros Mancoridis. Craft: A framework for evaluating software clustering results inthe absence of benchmark decompositions. In *Proceedings of the Eighth Working Conference on Reverse Engineering*, pages 93–103, October 2001.

[MM02] B. Mitchell and S. Mancoridis. Using heuristic search techniques to extract design abstractions from source code. In *Proc. Conf. on Genetic and evolutionary computation (GECCO)*, 2002.

[MM06] Brian S. Mitchell and Spiros Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[MM09] Ali Safari Mamaghani and Mohammad Reza Meybodi. Clustering of software systems using new hybrid algorithms. *Computer and Information Technology, International Conference on*, 1:20–25, 2009.

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

[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.

[MNS95] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 18–28, New York, NY, USA, 1995. ACM.

[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.

[MTW93] Hausi A. Müller, Scott R. Tilley, and Kenny Wong. Understanding software systems using reverse engineering technology perspectives from the rigi project. In *CASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*, pages 217–226. IBM Press, 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.

[Mye03] C. R. Myers. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical Review E*, 68(4):46116, 2003.

[PDP+07] Damien Pollet, Stephane Ducasse, Loic Poyet, Ilham Alloui, Sorana Cimpan, and Herve Verjus. Towards a process-oriented software architecture reconstruction taxonomy. In *Software Maintenance and Reengineering, 2007. CSMR '07. 11th European Conference on*, pages 137–148, 21-23 March 2007.

[PHLR09] Chiragkumar Patel, Abdelwahab Hamou-Lhadj, and Juergen Rilling. Software clustering using dynamic analysis and static dependencies. In *CSMR '09: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, pages 27–36, Washington, DC, USA, 2009. IEEE Computer Society.

[Puk94] Friedrich Pukelsheim. The three sigma rule. *The American Statistician*, 48(2):88–91, May 1994.

[Rag82] Vijay V. Raghavan. Approaches for measuring the stability of clustering methods. *SIGIR Forum*, 17(1):6–20, 1982.

[Rom90] H. C. Romesburg. *Clustering Analysis for Researchers*. Krieger, 1990.

[RoSU00] Andrew L. Rukhin, National Institute of Standards, and Technology (U.S.). *A statistical test suite for random and pseudorandom number generators for cryptographic applications [microform] / Andrew Rukhin ... [et al.]*. U.S. Dept. of Commerce, Technology Administration, National Institute of Standards and Technology ; For sale by the Supt. of Docs., U.S. G.P.O., Gaithersburg, MD : Washington, D.C. :, 2000.

[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.

[SBBP05] Olaf Seng, Markus Bauer, Matthias Biehl, and Gert Pache. Search-based improvement of subsystem decompositions. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1045–1051, New York, NY, USA, 2005. ACM Press.

[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.

[Sht] Mark Shtern. The factbase generator tool. Available online: `https://wiki.cse.yorku.ca/project/cluster/mdg_generator`.

[SMDM05] Ali Shokoufandeh, Spiros Mancoridis, Trip Denton, and Matthew Maycock. Spectral and meta-heuristic algorithms for software clustering. *J. Syst. Softw.*, 77(3):213–223, 2005.

[SMM02] A. Shokoufandeh, S. Mancoridis, and M. Maycock. Applying spectral methods to software clustering. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, page 3, Washington, DC, USA, 2002. IEEE Computer Society.

[Sof] Software Architecture Group (SWAG) at University of Waterloo. LSEdit: The graphical landscape editor. Available online: `http://www.swag.uwaterloo.ca/lsedit`.

[SS73] P. H. A. Sneath and R. R. Sokal. *Numerical Taxonomy: The Principles and Practice of Numerical Classification*. Series of books in biology, 1973.

[SS02] Eleni Stroulia and Tarja Systä. Dynamic analysis for reverse engineering and program understanding. *SIGAPP Appl. Comput. Rev.*, 10(1):8–17, 2002.

[ST04] Mark Shtern and Vassilios Tzerpos. A framework for the comparison of nested software decompositions. In *Proceedings of the Eleventh Working Conference on Reverse Engineering*, pages 284–292, November 2004.

[ST07] Mark Shtern and Vassilios Tzerpos. Lossless comparison of nested software decompositions. In *Reverse Engineering, 2007. Proceedings. 14th Working Conference on*, 2007.

[ST09a] Mark Shtern and Vassilios Tzerpos. Methods for selecting and improving software clustering algorithms. In *17th International Conference on Program Comprehension (ICPC 2009), May 17-19, 2009, Vancouver, British Columbia, Canada*, pages 248–252. IEEE Computer Society, 2009.

[ST09b] Mark Shtern and Vassilios Tzerpos. Refining clustering evaluation using structure indicators. In *ICSM '09: Proceedings of the 25st IEEE International Conference on Software Maintenance*, pages 297–305, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[ST10] Mark Shtern and Vassilios Tzerpos. On the comparability of software clustering algorithms. In *International Conference on Program Comprehension*, 2010.

[swaa] `http://www.swag.uwaterloo.ca/`.

[swab] `http://www.swat.uwaterloo.ca/~swagkit`.

[Tar00] Syst? Tarja. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. PhD thesis, Tampere University, 2000.

[TCLP04] Sander Tichelaar Timothy C. Lethbridge and Erhard Ploedereder. The dagstuhl middle metamodel: A schema for reverse engineering. In *Electronic Notes in Theoretical Computer Science*, volume 94, 2004.

[TGK99] Ladan Tahvildari, Richard Gregory, and Kostas Kontogianni. An approach for measuring software evolution using source code features. *apsec*, 00:10, 1999.

[TH97] V. Tzerpos and R.C. Holt. The orphan adoption problem in architecture maintenance. In *Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on*, pages 76–82, 6-8 Oct. 1997.

[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.

[TH00a] Vassilios Tzerpos and R. C. Holt. ACDC: An algorithm for comprehension-driven clustering. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 258–267, November 2000.

[TH00b] Vassilios Tzerpos and R. C. Holt. On the stability of software clustering algorithms. In *Proceedings of the Eighth International Workshop on Program Comprehension*, pages 211–218, June 2000.

[Tri01] Adrian Trifu. Using cluster analysis in the architecture recovery of object-oriented systems. Master's thesis, University of Karlsruhe, 2001.

[Tze01] Vassilios Tzerpos. *Comprehension-driven software clustering*. PhD thesis, University of Toronto, 2001.

[vL93] Gregor von Laszewski. A collection of graph partitioning algorithms:simulated annealing, simulated tempering, kemighan lin, two optimal, graph reduction, bisection. May 1993.

[VS] Sergi Valverde and Ricard V. Sole. Hierarchical small worlds in software architecture. Available online: `http://arxiv.org/abs/cond-mat/0307278`.

[WHH05] Jingwei Wu, Ahmed E. Hassan, and Richard C. Holt. Comparison of clustering algorithms in the context of software evolution. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 525–535, Washington, DC, USA, 2005. IEEE Computer Society.

[Wig97] T.A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on*, pages 33–43, 6-8 Oct. 1997.

[wik] `https://wiki.cse.yorku.ca/project/cluster`.

[Wis69] D. Wishart. Mode analysis: a generalization of nearest neighbour which reduces chaining effects. In Numerical Taxonomy, editor, *Numerical Taxonomy*, pages 282–3 11. Academic Press, 1969.

[WL07] Richard Wettel and Michele Lanza. Program comprehension through software habitability. In *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*, pages 231–240, 26-29 June 2007.

[WMFB+98] Robert J. Walker, Gail C. Murphy, Bjorn N. Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak. Visualizing dynamic software system information through high-level models. In *Conference on Object-Oriented*, pages 271–283, 1998.

[WS98] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, June 1998.

[WT03] Zhihua Wen and Vassilios Tzerpos. An optimal algorithm for MoJo distance. In *Proceedings of the Eleventh International Workshop on Program Comprehension*, pages 227–235, May 2003.

[WT04a] Zhihua Wen and V. Tzerpos. Evaluating similarity measures for software decompositions. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 368–377, 11-14 Sept. 2004.

[WT04b] Zhihua Wen and Vassilios Tzerpos. An effectiveness measure for software clustering algorithms. In *Proceedings of the Twelfth International Workshop on Program Comprehension*, pages 194–203, 2004.

[WT05] Z. Wen and V. Tzerpos. Software clustering based on omnipresent object detection. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, pages 269–278, 15-16 May 2005.

[Wuy01] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.

[Xan06] Spiros Xanthos. Clustering object-oriented software systems using spectral graph partitioning. 2006.

[Xia04] Chenchen Xiao. Using dynamic analysis to cluster large software systems. Master's thesis, York University, 2004.

[XLZS04] Xia Xu, Chung-Horng Lung, Marzia Zaman, and Anand Srinivasan. Program restructuring through clustering techniques. In *SCAM '04: Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop on (SCAM'04)*, pages 75–84, Washington, DC, USA, 2004. IEEE Computer Society.

[XT05] Chenchen Xiao and Vassilios Tzerpos. Software clustering based on dynamic dependencies. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 124–133, Washington, DC, USA, 2005. IEEE Computer Society.

[YGS+04] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman. Discotect: A system for discovering architectures from running systems, 2004.

[ZKS04] Shi Zhong, Taghi M. Khoshgoftaar, and Naeem Seliya. Analyzing software measurement data with clustering techniques. *IEEE Intelligent Systems*, 19(2):20–27, 2004.